Proving Weak Simulation via Strategy Synthesis

ANONYMOUS AUTHOR(S)

Simulation has been widely used to relate the behavior of two programs. A (strong) simulation relates a program state to another when any action executable from the first is available to be executed by the second and the resulting post states remain related. Weak simulation is defined similarly; however, it introduces a notion of observability (e.g., sending or receiving messages). While strong simulations preserve exact sequences of actions, weak simulation only requires preserving observationally equivalent sequences of actions. For many applications, strong simulation is not permissive enough. For example, consider two programs that both receive a key as input then look up in a hash table the value to output. If the two hash tables use different hash functions, then strong simulation would say the two programs are not equivalent. While under weak simulation the two programs would be equivalent (as internal computations are considered unobservable).

This paper introduces a method to automatically prove weak simulation between two integer message passing programs. Our technique is the first to automatically prove simulation (weak or otherwise) between two non-deterministic infinite-state programs. Our technique is a semi-algorithm that employs the game semantics of weak simulation to synthesize a (finite representation of a) strategy that witnesses the existence of a simulation.

ACM Reference Format:
Anonymous Author(s). 2024. Proving Weak Simulation via Strategy Synthesis. 1, 1 (February 2024), 36 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

There are many ways to define program equivalence including variants of Benton [2004]'s relational Hoare logic (RHL), trace equivalence, and variants of simulation [Milner 1971]. An important but challenging setting is message-passing programs, where we have to contend with (1) nondeterminism and (2) observability. Examples of this setting include distibuted, reactive, and real-tme systems and crypotographic protocols.

While variants of RHL and trace equivalence have seen extensive use in applications like translation validation, they are ill-suited to the context of message-passing programs. Consider the RHL specification $\{\bar{x} = \bar{x}'\}P \sim P'\{\bar{x} = \bar{x}'\}$, where P' is a copy of program P with each variable x replaced by a copy x'. The specification states that under any possible execution of P and P' if the programs start in identical states, then the programs end in idential states. Intuitively, one would expect this specification to always hold; however, if P is non-deterministic, then it may not. Additionally, it is possible for two programs P and Q to be trace equivalent even though Q may deadlock and P does not.

Milner [1989]'s work on simulation lays the foundation for defining program equivalence in the context of message-passing systems. In this paper, we consider a relational specification based on *divergence preserving weak simulation*. A classical (strong) simulation from program X to program Y requires that every behavior of X is matched step-by-step to a behavior of Y. In the context of message-passing programs (among others), simulation is not permissive enough. It is possible

XXXX-XXXX/2024/2-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

for programs X and Y to receive equal messages and respond with equal messages and yet X is not simulated by Y, as X and Y differ on the exact computations used to compute said responses. Weak simulation addresses this concern by relaxing the conditions for simulation to only consider observable behaviors—sending and receiving messages. However, weak simulation is too permisive, when X infinite loops, Y is free to do anything (e.g., terminate, infinite loop, or even send or receive messages). However, if the weak simulation relating X and Y is *divergence preserving* (X and Y have similar live-lock behaviors), Y must be able to infinite loop whenever X does.

We introduce a relational Hoare-style specification we call *contextual simulation* that takes the form $\{\mathcal{P}\}\ X \lesssim Y\ \{\mathcal{Q}\}$. Contextual simulation specifies that X must be related to Y by a divergence preserving weak simulation within the context of a pre-specification $\mathcal{P}$ and post-specification $\mathcal{Q}$.

In addition to defining contextual simulation, the main technical contribution of this paper is a semi-algorithm for proving or refuting the validity of a contextual simulation. While there are many techniques that can automatically compute simulation relations (of various kinds), our technique is the first to automatically prove the existence of a simulation (weak or otherwise) between two non-deterministic infinite state programs.

Verifying the validity of a contextual simulation $\{\mathcal{P}\}\ X \lesssim Y\ \{\mathcal{Q}\}$ comes with several challenges. It combines aspects of program safety verification (i.e all executions of X must be simulated by Y), termination verification (X and Y should be co-terminating), and program synthesis (non-determinism of Y should be treated angelically). In fact, the traditional Hoare logic partial correctness specification $\{P\}\ S\ \{Q\}$ can be encoded as the contextual simulation $\{P\}\ S \lesssim \texttt{while(}*\texttt{) skip}\ \{Q\}$, while the total correctness specification $[P]\ S\ [Q]$ can be encoded as $\{P\}\ S \lesssim \texttt{skip}\ \{Q\}$.

Our semi-algorithm, like several other methods for computing simulation [Bulychev et al. 2007; Etessami et al. 2005], is based on the game semantics of simulation. To prove or refute a contextual simulation $\{\mathcal{P}\}\ X \lesssim Y\ \{\mathcal{Q}\}$, we exhibit a (finite-representation of) a strategy for the induced simulation game. To compute such a strategy, we iteratively solve finite duration games (for the next n moves of the game). To solve finite duration games, the first step removes data non-determinism from Y by instantiating any non-deterministic term with a deterministic term. This process is similar to solving a sketch-based synthesis problem. Specifically, one could think of each non-deterministic term of Y as a hole and the task is to synthesize a (deterministic) term for each hole such that Y continues to simulate X. Afterwards, invaraint generation techniques are used to label the finite game's strategy with labels that prove Y continues to simulate X. Then, the labeled strategy is used to extend the overall strategy for a greater duration; however, only expanding is insufficient to handle programs with loops. At certain points, we check to see if the current state of the game to be expanded has already been expanded before. If so, the semi-algorithm tries to re-use the strategy previously expanded. If this would form a cycle in the strategy, we ensure that any fragments of X and Y contained within the cycle are co-terminating (cf. Secton 4 for full details).

The remainder of this paper is structured as follows. Section 2 provides background and defines contextual simulations. Section 3 gives the game semantics for contextual simulations. Section 4 describes our algorithm for synthesizing strategies for simulation games, which can be used to verify and refute contextual simulations. Section 5 describes SimVer, an implementation of our algorithm, and evaluates its performance. Section 6 compares our technique to related literature.

2 PRELIMINARIES

2.1 Programs

We consider simple message passing programs represented as control flow graphs.

Definition 2.1. A **Control Flow Graph** (CFG) is a finite labeled graph $G = \langle Loc, \dashrightarrow, in, out \rangle$, where:

- Loc is a finite set of control locations.
- $\dashrightarrow \subseteq Loc \times \texttt{com} \times Loc$ is a finite set of directed edges, each labeled by a *command*.
- $in \in Loc$ is a distinguished entry location
- $out \in Loc$ is a distinguished exit location with no outgoing edges.

The language of commands is as follows:

$$
\begin{aligned}
\langle com \rangle ::=\ & [\langle bexp \rangle] \mid \langle var \rangle := \langle exp \rangle \\
\mid\ & \text{havoc } \langle var \rangle .\ \langle bexp \rangle \\
\mid\ & \text{send } \langle exp \rangle \text{ chan}(\langle exp \rangle) \\
\mid\ & \text{receive } \langle var \rangle \text{ chan}(\langle exp \rangle)
\end{aligned}
\qquad
\begin{aligned}
\langle exp \rangle ::=\ & \langle var \rangle \mid c \in \mathbb{Z} \mid c \cdot \langle exp \rangle \mid \langle exp \rangle + \langle exp \rangle \\
\langle bexp \rangle ::=\ & \text{true} \mid \text{false} \mid \neg \langle bexp \rangle \mid \langle bexp \rangle \vee \langle bexp \rangle \\
\mid\ & \langle bexp \rangle \wedge \langle bexp \rangle \mid \langle bexp \rangle \leq \langle bexp \rangle
\end{aligned}
$$

The languages of expressions and Boolean expressions coincides with the languages of ground terms and formulas in linear integer arithmetic (LIA). In the remainder of the paper, we use "programs" and "control flow graphs" interchangeably. A program may include commands for guards (denoted as $[b]$ to mean `assume` b)), deterministic and non-deterministic assignments, and communication (using `send` and `receive`) along shared channels, which are identified by integers. Programs contain three forms of non-determinism: scheduler non-determinism arising from locations with multiple outgoing edges, message non-determinism (arising from `receive`), as well as the instruction `havoc x. b`, which non-deterministically assigns `x` a value such that `x` satisfies the Boolean expression `b`. For any instruction `receive x chan(c)`, we assume `x` does not appear in the expression `c`. CFGs may represent both sequential and concurrent programs. The standard construction of the CFG of two processes running concurrently is the Cartesian product of the CFG of the two concurrent processes.

Semantics. A **valuation** $\lambda : X \to \mathbb{Z}$ is a map from a finite set of variables X to the integers. We use $\lambda[x \mapsto v]$ to denote the valuation that maps x to v and every other variable y to $\lambda(y)$. For valuations λ_1 and λ_2 over disjoint domains, we use $\lambda_1 \uplus \lambda_2$ to denote their common extension. We use $[\![e]\!]_\lambda$ to denote the evaluation of a (Boolean) expression e under the valuation λ, assuming that the domain of λ contains the variables in e (with its usual interpretation).

Given a program, $P = \langle Loc, \dashrightarrow, in, out \rangle$ and a set of variables X, the semantics of P are defined by a labeled transition system $Trans(P) = \langle S, \longrightarrow, Init, Final \rangle$. The labels are drawn from an *observable alphabet* Σ and a single distinguished unobservable action which we denote by τ. Σ contains two types of actions: `send` actions of the form $\mathbf{s}(v, c)$ ("send v on channel c") and `receive` actions of the form $\mathbf{r}(v, c)$ ("receive v on channel c"), where v and c range over integers. A **program state** $\lambda \triangleright \ell$ is a valuation $\lambda : X \to \mathbb{Z}$ paired with a control location $\ell \in Loc$. S is the set of all such program states, $Init$ is the set of all initial states (where $\ell = in$), and $Final$ is the set of all final states (where $\ell = out$). Figure 1 gives the rules defining the transition relation $\longrightarrow$. Note that we use an open world assumption for communication: we suppose that the program is executed in an environment where external processes outside of the program can send and receive along any channel. Thus, communication instructions are never blocked, and a process may receive any value (including values that are not sent along that channel within the program); as a result, our semantics does not require an explicit representation of channel state. For brevity, for a program P, we will use Loc_P, $\dashrightarrow_P$, in_P, and out_P to refer to the components of P (i.e. $P = \langle Loc_P, \dashrightarrow_P, in_P, out_P \rangle$). Similarly, we use S_P, $\longrightarrow_P$, $Init_P$, and $Final_P$ to refer to the components of its transition system (i.e. $Trans(P) = \langle S_P, \longrightarrow_P, Init_P, Final_P \rangle$).

2.2 Simulation

We relate the behavior of two programs using *divergence preserving* [Van Glabbeek 2001] *weak simulations* [Milner 1989]. In a classical (strong) simulation, every transition of the system must be

$$\frac{\ell \overset{[b]}{\dashrightarrow} \ell' \qquad [\![b]\!]_\lambda}{\lambda \triangleright \ell \overset{\tau}{\longrightarrow} \lambda \triangleright \ell'} \qquad \frac{\ell \overset{x:=e}{\dashrightarrow} \ell' \qquad [\![e]\!]_\lambda = v}{\lambda \triangleright \ell \overset{\tau}{\longrightarrow} \lambda[x \mapsto v] \triangleright \ell'} \qquad \frac{\ell \overset{\mathsf{havoc}\ x.\ b}{\dashrightarrow} \ell' \qquad [\![b[x \mapsto v]]\!]_\lambda}{\lambda \triangleright \ell \overset{\tau}{\longrightarrow} \lambda[x \mapsto v] \triangleright \ell'}$$

$$\frac{\ell \overset{\mathsf{send}\ e\ \mathsf{chan}(ce)}{\dashrightarrow} \ell' \qquad [\![e]\!]_\lambda = v \qquad [\![ce]\!]_\lambda = c}{\lambda \triangleright \ell \overset{\mathbf{s}(v,c)}{\longrightarrow} \lambda \triangleright \ell'} \qquad \frac{\ell \overset{\mathsf{receive}\ x\ \mathsf{chan}(ce)}{\dashrightarrow} \ell' \qquad [\![ce]\!]_\lambda = c \qquad v \in \mathbb{Z}}{\lambda \triangleright \ell \overset{\mathbf{r}(v,c)}{\longrightarrow} \lambda[x \mapsto v] \triangleright \ell'}$$

Fig. 1. Transition rules for programs.

matched step by step with a transition of the protocol. Weak simulations relax this condition by matching every transition of the system with an *observationally equivalent* sequence of transitions. A simulation is divergence preserving if every divergent path (infinite sequence of unobservable transitions) of the system is matched by a divergent path of the protocol. To motivate our choice of simulation, consider the below schematic example implementation (left) and protocol (right), which differ in that the implementation includes some (communication-free) computation between receiving and sending a message.

Example 2.1.

```
while true do
  receive message;
  do_work();
  send response
done
```

```
while true do
  receive message;
  send response
done
```

Under strong simulation there is no possible simulation—the implementation takes more steps than the protocol. Under a *weak* simulation, there is a simulation even if "do_work" fails to terminate, which is undesirable because an important correctness property of the protocol—that every request is eventually serviced—is invalidated by the implementation. A *divergence preserving* weak simulation between the implementation and specification is only possible if "do_work" is terminating. In Theorem 2.4, we show that divergence preserving weak simulations preserve the universal fragment of action CTL* without next-time operators [Nicola and Vaandrager 1990].

First we give some auxiliary definitions. Given a program, P, a program state σ **silently reaches** program state σ' ($\sigma \overset{\tau}{\Longrightarrow}_P \sigma'$), when there is a (possibly empty) sequence of silent transitions from σ to σ'; that is, $\overset{\tau}{\Longrightarrow}_P \triangleq \overset{\tau}{\longrightarrow}_P^*$. For an observable action $\alpha \in \Sigma$, σ **α-observably reaches** σ' ($\sigma \overset{\alpha}{\Longrightarrow}_P \sigma'$), when there is a sequence of transitions from σ to σ', where one transition is an α transition and the rest are silent transitions; that is, $\overset{\alpha}{\Longrightarrow}_P \triangleq \overset{\tau}{\Longrightarrow}_P \circ \overset{\alpha}{\longrightarrow}_P \circ \overset{\tau}{\Longrightarrow}_P$.

Definition 2.2 (Weak Simulation). A binary relation $R \subseteq S_P \times S_Q$ from program states of P to program states of Q is a **weak simulation** (from P to Q), if for any pair of states σ_P and σ_Q related by R (written $\sigma_P R \sigma_Q$), and all $\sigma'_P \in S_P$ and $\alpha \in \Sigma \cup \{\tau\}$ such that $\sigma_P \overset{\alpha}{\longrightarrow}_P \sigma'_P$, there exists some $\sigma'_Q \in S_Q$ such that $\sigma_Q \overset{\alpha}{\Longrightarrow}_P \sigma'_Q$ and $\sigma'_P R \sigma'_Q$.

Definition 2.3 (Divergence Preserving). A weak simulation, $R \subseteq S_P \times S_Q$, from program P to program Q is divergence preserving if for any pair of states σ_P and σ_Q related by R ($\sigma_P R \sigma_Q$), and all infinite silent paths, $\sigma_{P0} \overset{\tau}{\longrightarrow}_P \sigma_{P1} \overset{\tau}{\longrightarrow}_P \ldots$, starting from σ_P ($\sigma_P = \sigma_{P0}$), there exists an infinite silent path $\sigma_{Q_0} \overset{\tau}{\longrightarrow} \sigma_{Q_1} \overset{\tau}{\longrightarrow}_Q \ldots$ starting from σ_Q ($\sigma_Q = \sigma_{Q_0}$) and an infinite sequence $k_1, k_2, \ldots$ of naturals such that σ_{P_i} is R-related to $\sigma_{Q_{k_i}}$ for all i, and which is ascending ($k_i \leq k_{i+1}$ for all i) and unbounded (for all $n \in \mathbb{N}$, there is some i such that $k_i > n$).

It is well known that simulation relations preserve temporal logic formulas [Bensalem et al. 1992; Bulychev et al. 2007; Parrow et al. 2017]. We show that every divergence preserving weak simulation preserves the universal fragment of action CTL* without next time operators. Action CTL^* is a branching-time logic for reasoning about labeled transition systems with observable actions [Nicola and Vaandrager 1990]. For example, action CTL^* is able to formalize specifications such as "every request eventually receives a response" and "eventually every node will respond with the same value." We provide full details of our formalization in the proof of Theorem 2.4 in Appendix C.

Theorem 2.4. *Let φ be any formula of the universal fragment of action CTL^* without next-time operators ($\forall ACTL^*$-$\{X_p, X_\tau\}$). If program P is related to program Q by a divergence preserving weak simulation and Q satisfies φ then P satisfies φ.*

See proof on page 30.

Contextual simulations are modular specifications of correctness of message passing programs based on divergence preserving weak simulation. A *contextual simulation* is a quadruple $\{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$ where *src* and *tgt* are both programs (presumed to be operating over disjoint sets of variables, say X and Y), and $\mathcal{P}$ and $\mathcal{Q}$ are Boolean expressions ranging over variables of both *src* and *tgt*. We call *src* the *source* or *implementation* program and *tgt* the *target* or *specification* program. Since *src* and *tgt* operate over disjoint variables, we may use ordinary first-order formulas over both sets of variables as predicates for these joint states. Intuitively, $\{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$ asserts that any pair consisting of a *src*-state and *tgt*-state that jointly satisfy $\mathcal{P}$ are observationally equivalent and (after executing *src* and *tgt*) end in states that are related by $\mathcal{Q}$. For example, contextual simulations can express that a distributed system only implements the protocol when started in equivalent states, or that when both the implementation and protocol terminate they do so in related states.

Definition 2.5 (Contextual Simulation). We say a **contextual simulation** holds, $\models \{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$, if there exists a divergence preserving weak simulation $R \subseteq S_{src} \times S_{tgt}$ such that

- R respects the pre-condition $\mathcal{P}$: every initial state of *src* and *tgt* that jointly satisfies $\mathcal{P}$ is related by R. That is, for all $\lambda_{src} : X \to \mathbb{Z}, \lambda_{tgt} : Y \to \mathbb{Z}$ such that $[\![\mathcal{P}]\!]_{\lambda_{src} \uplus \lambda_{tgt}}$ is true, we have $(\lambda_{src} \triangleright in_{src}) R (\lambda_{tgt} \triangleright in_{tgt})$.
- R respects the post-condition $\mathcal{Q}$: whenever a final state σ_{src} is related to a state σ_{tgt} by R, then σ_{tgt} must silently reach a final state σ'_{tgt} such that σ_{src} and σ'_{tgt} jointly satisfy the post-condition $\mathcal{Q}$. That is, for all $(\lambda_{src} \triangleright out_{src})$ and σ_{tgt} such that $(\lambda_{src} \triangleright out_{src}) R \sigma_{tgt}$, there is some λ_{tgt} such that $\sigma_{tgt} \overset{\tau}{\Longrightarrow}_{tgt} (\lambda_{tgt} \triangleright out_{tgt})$ and $[\![\mathcal{Q}]\!]_{\lambda_{src} \uplus \lambda_{tgt}}$ is true.

3 GAME SEMANTICS OF SIMULATION

This section describes (1) a game semantics for contextual simulations and (2) *labeled simulation game unwindings*, a finite representation of a (partial) strategy for Verifier in a simulation game. This forms the basis of the algorithm in Section 4 for verifying contextual simulations. Figure 2 displays a contextual simulation along with a complete well-labeled game unwinding, which we will use as a running example.

3.1 Semantic Simulation Game

Every contextual simulation defines an infinite game $\mathcal{G}(\{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\})$ played by two players, Falsifier and Verifier. Verifier's goal is to prove the validity of the contextual simulation, Falsifier's is to disprove it. If we look at Definition 2.2, we see that each step of the implementation must be matched by an observably equivalent sequence of transitions from the specification. In our game, Falsifier controls the implementation and Verifier the Specification. Intuitively, in a play of the game, Falsifier tries to construct a trace of the implementation that has no observationally equivalent trace in the Specification, whereas Verifier tries to construct an observationally equivalent trace of the specification for the trace Falsifier constructs.

A play of the game proceeds with Falsifier and Verifier taking turns choosing *moves* forever (not necessarily strictly alternating between the two players). A move consists of the active player choosing the next *place*. A place is either a *Falsifier place* or *Verifier place*. A Falsifier place dictates that the next move belongs to Falsifier, while Verifier places dictate that the next move belongs to Verifier. A **Falsifier place** takes the form $F\left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ where $\ell_{src} \in Loc_{src}$, $\ell_{tgt} \in Loc_{tgt}$, and $\lambda : (X \cup Y) \to \mathbb{Z}$ (recall we assume *src* and *tgt* operate over disjoint sets of variables, X and Y). A **Verifier place** takes the form $V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ where $\alpha \in \Sigma \cup \{\tau\}$ indicates the most recent label a transition executed by *src*, and ℓ_{src}, ℓ_{tgt}, and λ are as before.

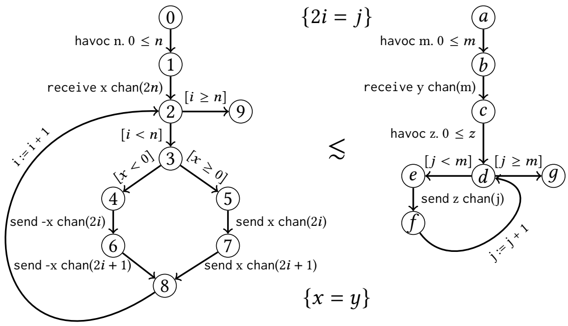
The set of all moves M is the set of all Verifier and Falsifier places. A **position** $s \in M^*$ is a finite sequence of moves, and a **play** $p \in M^\omega$ is an infinite sequence of moves. Falsifier makes the first move. Afterwards, the next player to make a move is dictated by the final place of the position (e.g. Falsifier makes the next move if and only if the final place of the position is a Falsifier place). We define the winning conditions in terms of the legal positions of the game. The legal positions are defined inductively as follows:

- (Initialization) The game begins in an arbitrary joint state λ satisfying the pre-condition $\mathcal{P}$, with the source and target in their initial positions and with Falsifier to play. Formally:
  If $[\![\mathcal{P}]\!]_\lambda$ is true, then $F\left\langle in_{src}, in_{tgt}, \lambda \right\rangle$ is legal.
- (Falsifier) For a legal prefix ending in a Falsifier place, the game continues where Falsifier must choose an outgoing transition of *src* and let Verifier attempt to match the chosen transition. Formally:
  If $s \cdot F\left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ is legal and $\lambda \triangleright \ell_{src} \xrightarrow{\alpha}_{src} \lambda' \triangleright \ell'_{src}$, then $s \cdot F\left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle \cdot V\left\langle \alpha, \ell'_{src}, \ell_{tgt}, \lambda' \right\rangle$ is legal
- (Match) For a legal prefix ending in a Verifier place, Verifier may continue the game by choosing an outgoing transition of *tgt* that is labeled with the same action of the transition previously chosen by Falsifier. Verifier then continues its turn released of its obligation of executing a transition matching Falsifer's action. Formally:
  If $s \cdot V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ is legal and $\lambda \triangleright \ell_{tgt} \xrightarrow{\alpha}_{tgt} \lambda' \triangleright \ell'_{tgt}$, then $s \cdot V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle \cdot V\left\langle \tau, \ell_{src}, \ell'_{tgt}, \lambda' \right\rangle$ is legal
- (Continue) For a legal prefix ending in a Verifier place, Verifier may continue the game by choosing a silent transition of *tgt*. Verifier then continues its turn. Formally:
  If $s \cdot V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ is legal and $\lambda \triangleright \ell_{tgt} \xrightarrow{\tau}_{tgt} \lambda' \triangleright \ell'_{tgt}$, then $s \cdot V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle \cdot V\left\langle \alpha, \ell_{src}, \ell'_{tgt}, \lambda' \right\rangle$ is legal
- (Pass) For a legal prefix ending in a Verifier place, if Verifier has satisfied its matching obligation then Verifier may choose to pass their turn. Formally:
  If $s \cdot V\left\langle \tau, \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ is legal, then $s \cdot V\left\langle \tau, \ell_{src}, \ell_{tgt}, \lambda \right\rangle \cdot F\left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ is legal

We say that Falsifier wins a play if

- There is an illegal prefix of the form $s \cdot V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle \cdot m$ such that $s \cdot V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ is legal (i.e., Verifier makes the first illegal move), or
- There is a legal prefix of the form $s \cdot V\left\langle \tau, out_{src}, \ell_{tgt}, \lambda \right\rangle \cdot F\left\langle \tau, out_{src}, \ell_{tgt}, \lambda \right\rangle$ where $[\![Q]\!]_\lambda$ is false or $\ell_{tgt} \neq out_{tgt}$, or
- Every prefix is legal and Verifier either always passes or always continues.

A **strategy** for Verifier is a function $g : \{V\left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle \in M\} \to M$ that maps each Verifier place to a move. We say a play $p = p_0 p_1 p_2 \ldots$ **conforms** to Verifier's strategy, g, when every move



(A) Contextual Simulation $\{2i = j\}$ *src* $\lesssim$ *tgt* $\{x = y\}$

$\varphi_1 \triangleq 2i = j \land 0 \leq n$ $\quad$ $\varphi_2 \triangleq 2i = j \land 2n = m$

$\varphi_3 \triangleq \varphi_2 \land x = y$ $\quad$ $\varphi_4 \triangleq \varphi_3 \land x < 0 \land z = -x$

$\varphi_5 \triangleq \varphi_3 \land x \geq 0 \land z = x$ $\quad$ $\varphi_6 \triangleq \varphi_3 \land i \geq n$

$\varphi_7 \triangleq \varphi_4 \land i < n$ $\quad$ $\varphi_8 \triangleq \varphi_6 \land i < n$

$\varphi_9 \triangleq 2i + 1 = j \land 2n = m \land x = y \land x < 0 \land z = -x \land i < n$

$\varphi_{10} \triangleq 2i + 1 = j \land 2n = m \land x = y \land x \geq 0 \land z = x \land i < n$

$\varphi_{11} \triangleq 2i = j + 1 \land 2n = m \land x = y \land x < 0 \land z = -x$

$\varphi_{12} \triangleq 2i = j + 1 \land 2n = m \land x = y \land x \geq 0 \land z = x$

(B) Formula annotations for labeled unwinding

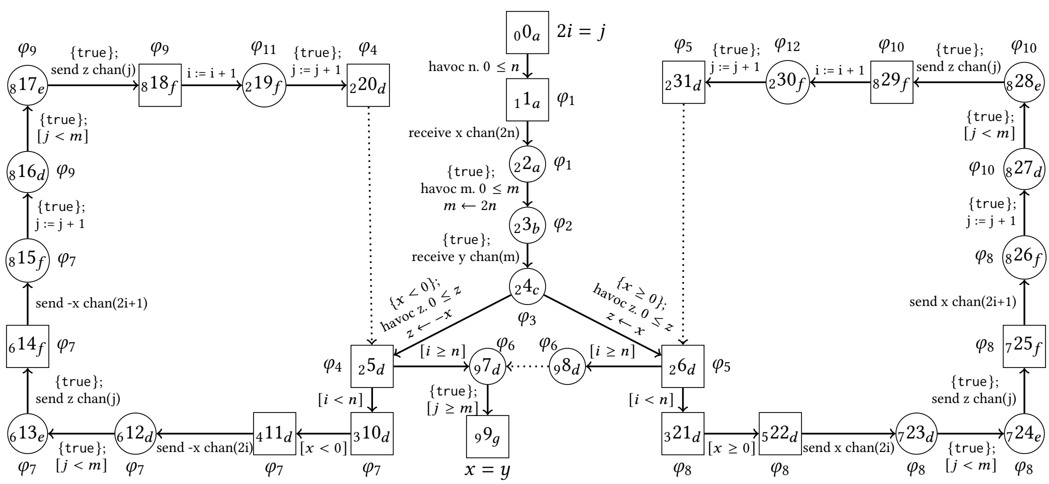


Fig. 2. A complete, well-labeled Simulation Game Unwinding

made by Verifier is decided by g; that is, for all i such that p_i is a Verifier place, we have $p_{i+1} = g(p_i)$. We say g is **winning** if every play that conforms to g is won by Verifier. Strategies for Falsifier are defined analogously.

Theorem 3.1. *The contextual simulation $\{\mathcal{P}\}$ src $\lesssim$ tgt $\{\mathcal{Q}\}$ is valid if and only if Verifier has a winning strategy for $\mathcal{G}(\{\mathcal{P}\}$ src $\lesssim$ tgt $\{\mathcal{Q}\})$.*

See proof on page 33.

3.2 Simulation Game Unwindings

We propose simulation game unwindings as a finite representation of strategies for Verifier. Simulation game unwindings are proof objects that certify a given contextual simulation (analogous to program unwindings in program verification [McMillan 2006]). Figure 2 displays an example contextual simulation and a complete well-labeled simulation game unwinding proving its validity. We first give intuition into how the example game unwinding corresponds to a strategy for Verifier, then give the definition of (un)labeled simulation game unwindings, and finish by defining when a simulation game unwinding is well-labeled and complete.

Figure 2 can be understood as a representation of an infinite set of legal plays of the simulation game that conform to a strategy. Each node represents (a set of) places which belong to either Falsifier (square) or Verifier (circle). Any legal play starts at the root node ${}_0 0_a$ (indicating a Falsifier place). In the next move, Falsifier executes its havoc action and Verifier responds by passing its turn (indicated by the fact that the edge ${}_00_a \to {}_11_a$ connects two Falsifier nodes). Falsifier receives a message, to which Verifier responds by first playing its `havoc m` command, choosing the value $2n$ for m (indicated by the edge label $m \leftarrow 2n$), receiving a message, and then playing its `havoc z` command, and passing the turn back to Falsifier. If the value received by the source program is negative, then z is set to $-x$ (indicated by the left edge outgoing from ${}_24_c$), and if the value is non-negative, then z is set to x (the right edge). In either case, Falsifier begins at the loop header of the source program and either enters the loop body or exits the program. The process continues analogously.

Definition 3.2 (Simulation Game Unwinding). A **Simulation Game Unwinding** from program *src* to program *tgt* is a finite bipartite tree $U = \langle F, V, E, r, L, S, T \rangle$, where:

- F and V are finite disjoint sets of nodes. Define $N \triangleq F \cup V$ to be the set of all nodes
- $\langle N, E \rangle$ is a finite tree rooted at $r \in F$
- $S : N \to Loc_{src}$ and $T : N \to Loc_{tgt}$ map each node to a *src* and *tgt* control location, respectively
- $L : E \to com$ maps each edge to a command

such that $S(r) = in_{src}$ and $T(r) = in_{tgt}$, and for each edge $\langle u, v \rangle \in E$:

- If $u \in F$, then $S(u) \overset{L(u,v)}{\dashrightarrow}_{src} S(v)$ and $T(u) = T(v)$, and
- If $u \in V$, then $T(u) \overset{L(u,v)}{\dashrightarrow}_{tgt} T(v)$ and $S(u) = S(v)$.

In Figure 2, each node is given the label ${}_{S(n)}n_{T(n)}$. We represent F-nodes with squares (e.g. ${}_00_a$) and V-nodes with circles (e.g. ${}_22_a$). Every edge $\langle u, v \rangle$ is labeled with the command $L(u, v)$. A simulation game unwinding from *src* to *tgt* represents a joint-unwinding of the two programs starting from the initial location of both programs. Each F-node unwinds one step of *src*, while each V-node unwinds one step of *tgt*. For instance, we see that from node 0 there is a single edge to node 1 labeled with the first command executed by *src*. Similarly, node 2 has a single edge to node 3 labeled with the first command executed by *tgt*.

Definition 3.3 (Labeled Simulation Game Unwinding). A **Labeled Simulation Game Unwinding** (from *src* to *tgt*) is a tuple $\mathcal{L} = \langle U, \Phi, K, G, X, \triangleright, m \rangle$, where

- $U = \langle F, V, E, r, L, S, T \rangle$ is a simulation game unwinding from *src* to *tgt*.
- $\Phi : N \to bexp$ is a *vertex label* mapping each node to a formula over the variables of both *src* and *tgt*.
- $K : E \nrightarrow exp$ is a partial map, which maps each edge $\langle u, v \rangle$, where $u \in V$ and $L(u, v)$ is a `havoc`, to an expression that determinizes the havoc command.
- $G : E \nrightarrow bexp$ is a partial map, which maps each edge $\langle u, v \rangle$ where $u \in V$ to a *guard*, a formula encoding the condition when the edge is taken.
- $X \subseteq N$ is a set of *expanded* nodes; nodes in $N \setminus X$ are leaves of the tree.
- $\triangleright \subseteq (N \setminus X) \times X$ is a *covering relation*, with $u \triangleright v$ indicating that the state of the simulation game represented by u is subsumed by the state of the game at v. F nodes may only be covered by F-nodes, and V nodes may only be covered by V-nodes (i.e. if $u \triangleright v$ then $u \in F \Rightarrow v \in F$ and $u \in V \Rightarrow v \in V$).
- m_F and m_V are *measures* (ranking functions), which serve as witnesses to certain well-foundedness conditions to be described in the following. The well-foundedness conditions ensure that Verifier may neither pass forever nor continue forever.

In Figure 2, next to each node, n, we display the formula $\Phi(n)$ (e.g. $\Phi(0)$ is $2i = j$). Each V-edge, $\langle u, v \rangle$, $(u \in V)$ is labeled with a guard (displayed as $\{G(u, v)\}$). Additionally, each V-edge from u to v labeled with a havoc command ($L(u, v)$ is some `havoc x. b`) is labeled with a term $K(u, v)$ (displayed as $x \leftarrow K(u, v)$). We display each $u \triangleright v$ as a dotted edge from node u to node v. X is primarily a book-keeping variable and does not have a graphical representation. We also omit the measures m_F and m_V from Figure 2, since the well-foundedness conditions are trivial for this example.

Each labeled unwinding, $\mathcal{L}$, represents a Verifier strategy, $g_{\mathcal{L}}$. To define $g_{\mathcal{L}}$, we first associate each node n with a set of places $Places(n)$, represented by the node's labels. If n is an F-node, then n is associated with all Falsifier places of the form $F\,\langle S(n), T(n), \lambda\rangle$, where $[\![\Phi(n)]\!]_\lambda$ is true. If n is a V-node, then n is associated with Verifier places of the form $V\,\langle \alpha, S(n), T(n), \lambda\rangle$ where α is the action to be matched, and $[\![\Phi(n)]\!]_\lambda$ is true. (Note: we can compute α by looking at the path from the root of the unwinding to n.)

For example in Figure 2, $Places(3)$ contains all places $V\,\langle \mathbf{r}(msg, c), 2, b, \lambda\rangle$, where $[\![\varphi_2]\!]_\lambda$ is true, $msg = [\![x]\!]_\lambda$ and $c = [\![2n]\!]_\lambda$, since the preceding F-edge from node 1 to node 2 is labeled with `receive x chan(2n)`. $Places(4)$ contains all places $V\,\langle \tau, 2, c, \lambda\rangle$, where $[\![\varphi_3]\!]_\lambda$ is true, because the `receive` command from node 1 to node 2 was already matched by the V-edge from node 3 to node 4.

Given $Places$ we can define $g_{\mathcal{L}}$: given any Verifier place, $V\,\langle \alpha, \ell_s, \ell_t, \lambda\rangle$, if $V\,\langle \alpha, \ell_s, \ell_t, \lambda\rangle$ belongs to $Places(n)$ for some expanded node n, then Verifier chooses a successor n' of n, whose guard is satisfied by λ and plays according to that edge. If $V\,\langle \alpha, \ell_s, \ell_t, \lambda\rangle$ does *not* belong to $Places(n)$ for any n, Verifier passes the turn.

Consider the place $V\,\langle \tau, 2, c, \lambda\rangle$, where $[\![\varphi_3]\!]_\lambda$ is true. This place is associated with node 4. Necessarily, λ either satisfies $x < 0$ (the guard from node 4 to node 5) or $x \geq 0$ (the guard from node 4 to node 6). In the first case, we see the edge from node 4 to 5 is labeled with `havoc z.` $0 \leq z$ and a term $-x$ that represents Verifier's chosen strategy (z is assigned the value of $-x$). In this case, the next move is $V\,\langle \tau, 2, d, \lambda[z \mapsto c]\rangle$ where $c = [\![-x]\!]_\lambda$. The process proceeds analogously for all Verifier places. We show in Theorem 3.6, that if the unwinding is *well-labeled* and *complete*, then $g_{\mathcal{L}}$ is a winning strategy for Verifier.

Definition 3.4 (Complete). A labeled simulation game unwinding, $\mathcal{L} = \langle U, \Phi, K, G, X, \triangleright, m\rangle$, is **complete** when every node ($u \in N$) is either expanded ($u \in X$) or covered ($\exists v. u \triangleright v$).

The simulation game unwinding in Figure 2 is complete. The only un-expanded nodes are 8, 20, and 31, which are covered by nodes 7, 5, and 6 respectively. The unwinding is also well-labeled, as we will now define.

Given a labeled unwinding $\mathcal{L}$, for any node $v \in N$, there is unique tree path $v_0 v_1 ... v_n$ from the root to v (i.e., $r = v_0$, $v_n = v$, and $\langle v_i, v_{i+1}\rangle \in E$ for all i). Define $F\text{-}pred(v)$ to be v_i, where i is the greatest index such that $v_i \in F$, and define $F\text{-}pred_e(v) \triangleq L(v_i, v_{i+1})$ to be the command labeling the edge leaving $F\text{-}pred(v)$. $F\text{-}pred(v)$ is well-defined for all nodes (in particular, $F\text{-}pred(v) = v$ for all $v \in F$). $F\text{-}pred_e(v)$ is defined only if $v \in V$.

Figure 3 defines two auxiliary functions on edges: *legal*, which represents when the given V-edge is allowed to be played; and *act*, which represents how the post-state (primed variables) is related to the pre-state (un-primed variables) when taking the given edge. Note that *legal* and *act* (1) determinize V-edge `havoc` commands (using the K map) and (2) encode when V-edge `send` (resp. `receive`) commands are legal (if the preceding F-edge is labeled with a `send` (resp. `receive`) command, then equal messages are sent (resp. received) along equal channels).

Definition 3.5 (Well-Labeled). A labeled simulation game unwinding, $\mathcal{L} = \langle U, \Phi, K, G, X, \triangleright, m\rangle$, is **well-labeled** for a contextual simulation, $\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$, when the Initial, Final, Consecution, Observational Matching, Covering, Well-foundedness, and Adequacy conditions are met.

Initial: The root node annotation is entailed by the precondition: $\mathcal{P} \models \Phi(r)$.

$$
legal(u,v) \triangleq \begin{cases} b & \text{if } L(u,v) = [b] \\ b[x \mapsto K(u,v)] & \text{if } L(u,v) = \text{havoc } x.\ b \\ e_s = e_t \wedge c_s = c_t & \text{if } \begin{array}{l} L(u,v) = \text{send } e_t \text{ chan}(c_t) \text{ and} \\ F\text{-}pred_e(u) = \text{send } e_s \text{ chan}(c_s) \end{array} \\ c_s = c_t & \text{if } \begin{array}{l} L(u,v) = \text{receive } x \text{ chan}(c_t) \text{ and} \\ F\text{-}pred_e(u) = \text{receive } y \text{ chan}(c_s) \end{array} \\ false & \text{if } L(u,v) \text{ is observable} \\ true & \text{otherwise} \end{cases}
\qquad
act(u,v) \triangleq \begin{cases} x' = e \wedge \bigwedge_{y \neq x} y = y' & \text{if } L(u,v) = x := e \\ x' = K(u,v) \wedge \bigwedge_{y \neq x} y = y' & \text{if } L(u,v) = \text{havoc } x.\ b \text{ and } u \in V \\ b[x \mapsto x'] \wedge \bigwedge_{y \neq x} y = y' & \text{if } L(u,v) = \text{havoc } x.\ b \text{ and } u \in F \\ x' = y' \wedge \bigwedge_{z \neq x} z = z' & \text{if } \begin{array}{c} L(u,v) = \text{receive } x \text{ chan}(c_t) \\ \text{and } u \in V \text{ and} \\ F\text{-}pred_e(u) = \text{receive } y \text{ chan}(c_s) \end{array} \\ \bigwedge_{y \neq x} y = y' & \text{if } \begin{array}{c} L(u,v) = \texttt{receive } x \texttt{ chan}(c) \\ \text{and } u \in F \end{array} \\ b \wedge \bigwedge_{x \in X} x = x' & \text{if } L(u,v) = [b] \text{ and } u \in F \\ \bigwedge_{x \in X} x = x' & \text{otherwise} \end{cases}
$$

Fig. 3. Determines the legality and action of an edge of the game unwinding.

The initial condition ensures that every initial state of the source program and target program related by the pre-condition $\mathcal{P}$ are related by the annotations of the root node. We can verify that this condition holds for the labeled unwinding in Figure 2: the root node's annotation is $2i = j$, which is exactly the given pre-condition.

Final: Every final node must have a label strong enough to prove the required post-condition Q:

$$\forall u \in X.\ S(u) = out_{src} \wedge T(u) = out_{tgt} \Rightarrow \Phi(u) \models Q$$

The final condition ensures that when both the source and target program reach a final state, they jointly satisfy the post-condition Q. We see that the labeled unwinding in Figure 2 has only one final node (node 9) and its annotation $x = y$ is exactly the required post-condition.

Consecution: Each edge $\langle u, v \rangle \in E$ must satisfy both of the following conditions.

(1) If $u \in F$, then $\Phi(u)(X) \wedge act(u,v)(X, X') \models \Phi(v)(X')$
(2) If $u \in V$, then $\Phi(u)(X) \wedge G(u,v)(X) \wedge act(u,v)(X, X') \models legal(u,v)(X) \wedge \Phi(v)(X')$

The first rule ensures that if Falsifier has a legal response m' following the edge from u to v to some place in $Places(u)$, then m' belongs to $Places(v)$. The second rule ensures that for any place m in $Places(u)$ such that the valuation of m satisfies the guard $G(u,v)$, Verifier has a *legal* move m' (executing the command $L(u,v)$, treating `havoc x. b` as an assignment of $K(u,v)$ to x) such that $m' \in Places(v)$.

For example, for any place associated to node 4, the valuation either satisfies the guard from node 4 to node 5 ($x < 0$) or the guard from node 4 to node 6 ($x \geq 0$). If it satisfies the guard from 4 to 5, the second consecution rule ensures that there is a legal move associated with node 5 that is decided by K. It holds analogously, if the valuation satisfies the guard from 4 to 6.

Observational Matching: Every send (resp. receive) command labeling an edge outgoing from an F-node must eventually be matched by a send (resp. receive) command labeling an edge outgoing from a V-node along every path starting from the F-node's send (resp. receive) command. More formally, for any F-edge, $\langle u, v \rangle \in E$ and $u \in F$ such that $L(u,v)$ is a send (resp. receive) command, then for each path $p = v_0, \ldots, v_n$ from $v = v_0$ to a leaf node v_n there is a unique v_i such that u is its most recent F-ancestor (F-$pred(v_i) = u$), and either v_i is v_n and is un-expanded ($v_i = v_n \notin X$) or $L(v_i, v_{i+1})$ is a send (resp. receive) command.

This rule ensures the syntactic requirements that every F-edge labeled with a send (resp. receive) command is always eventually matched by a single send (resp. receive) command labeling a V-edge before the next F-node is encountered. That is, when Verifier ends their turn, it is legal for Verifier to do so. For example, in Figure 2, we see that the receive command along the edge from 1 to 2 is followed by a receive command along the edge from 3 to 4 and the only edges between the two and between nodes 4 and 5 and 4 and 6—Verifier ends their turn at nodes 5 and 6—are labeled with havocs (unobservable) commands.

Covering: If $u \triangleright v$ then $\Phi(u) \models \Phi(v)$, $S(u) = S(v)$, $T(u) = T(v)$. Moreover, if $u \in V$, then either both $F\text{-}pred_e(u)$ and $F\text{-}pred_e(v)$ are unobservable or $F\text{-}pred_e(u) = F\text{-}pred_e(v)$.

In order to cover a node (closing the path it ends), we must ensure there is an expanded node covering it such that any move associated with the covered node is also associated with the covering node. In Figure 2, nodes 8, 20, and 31 are covered by 7, 5, and 6 respectively.

Well-foundedness: We require that every path consisting of both tree edges and covering edges is either finite or visits both F-nodes and V-nodes infinitely often. To ensure this property, we require that the measure m_F (mapping state pairs to some well-founded order) is positive and strictly decreasing on every edge outgoing from an F-node. Similarly, m_V must be positive and strictly decreasing on every edge outgoing from a V-node.

The condition on m_V ensures that the Verifier strategy associated with a well-labeled unwinding always passes the turn to Falsifier after a finite number of steps. The condition on m_F ensures that the produced strategy is also divergent preserving. These conditions rule out cases where the target (resp. source) program has an infinite cycle containing only unobservable actions (and is not matched by an equi-terminating unobservable cycle within the source (reps. target) program). While m_V tends to rule out a pathological case—the protocol contains a silent loop—, m_F is often important in distributed systems, since application logic may introduce loopy (but terminating) computations on messages received or messages to be sent. Recall Example 2.1, m_F ensures that we only allow proving simulation if "do_work" is terminating. All of the loops in Figure 2 contain both V-nodes and F-nodes, and so explicit measures are not necessary.

Adequacy: For each expanded node $u \in X$,

(1) If $u \in F$, then for each $S(u) \overset{a}{\dashrightarrow} l_s$ such that a is consistent with $\Phi(u)$, there must be some node v such that $\langle u, v \rangle \in E$, $L(u, v) = a$, and $S(v) = l_s$.
(2) If $u \in V$, then $\Phi(u) \models \bigvee_{\langle u,v \rangle \in E} G(u, v)$.

The first adequacy condition ensures that if Falsifier has a legal response m' to a place $m \in Places(u)$ then there is some successor v of u with $m' \in Places(v)$. The second ensures that starting from any place $m \in Places(u)$, Verifier has some response to make (i.e. m's valuation satisfies at least one guard labeling the outgoing edges of u). In Figure 2, we can verify both of these conditions for every node. While nodes 10 and 21 of the labeled unwinding are at location 4 of the source program, which is a branching point of the CFG we see only one outgoing edge for either node. This is because the annotations of 10 and 21 are sufficient to rule out the other branch of the CFG. Similarly, most V-nodes trivially satisfy the second condition as they only have one successor node and the strategy guard of the outgoing edge is true. The interesting case is node 4. We see one outgoing edge guarded by $x < 0$ and the other guarded by $0 \leq x$, which together cover φ_3 (the label of node 4).

Theorem 3.6. *If there is a well-labeled complete simulation game tree for $\{\mathcal{P}\}$ src $\lesssim$ tgt $\{Q\}$, then Verifier has a winning strategy for $\mathcal{G}(\{\mathcal{P}\}$ src $\lesssim$ tgt $\{Q\})$.*

See proof on page 34.

4 SIMULATION VERIFICATION

This section presents an algorithm, Algorithm 1, for verification and refutation of contextual simulations. The algorithm is based on the game semantics for contextual simulation (Section 3). Given a contextual simulation, $\{\mathcal{P}\}$ *src* $\lesssim$ *tgt* $\{Q\}$, the algorithm either (1) constructs a complete well-labeled game unwinding (a strategy for Verifier, which serves as a proof of the validity of the contextual simulation), (2) constructs a winning strategy for Falsifier for a finite unrolling of the game (a refutation of the contextual simulation), or (3) runs forever.

Algorithm 1 is inspired by Farzan and Kincaid [2017]'s method for synthesizing strategies for safety games. It maintains a well-labeled simulation game unwinding $\mathcal{L}$, which is initialized to contain just the root node r. If at any step $\mathcal{L}$ is complete, then Verifier has a winning strategy and the contextual simulation is valid. Otherwise, there is a witness to failure of the completeness condition: a node v of $\mathcal{L}$ that is neither expanded nor covered. The algorithm proceeds by finding a node to cover v, or (failing that) expanding the node v by computing a winning strategy for either Verifier or Falsifier for a finite-horizon of the game. If Verifier wins the finite-horizon game, the algorithm uses Verifier's winning strategy to expand v; if Falsifier wins, it backtracks and expands v's parent with a greater horizon. If Falsifier wins a finite-horizon game starting from the root, then the contextual simulation is refuted.

Algorithm 1: Strategy synthesis for contextual simulation.

Procedure Strategy-synthesis($\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$)
- $r \leftarrow$ fresh vertex, $F \leftarrow \{r\}$, $V \leftarrow \emptyset$, $\Phi(r) \leftarrow \mathcal{P}$;
- $S(r) \leftarrow in_{src}$, $T(r) \leftarrow in_{tgt}$, $E \leftarrow \emptyset$, $L \leftarrow \emptyset$;
- $X \leftarrow \emptyset$, $K \leftarrow \emptyset$, $G \leftarrow \emptyset$, $\triangleright \leftarrow \emptyset$;
- **while** *$\mathcal{L}$ is not complete* **do**
  - Pick any $v \in N \setminus X$ that is not covered;
  - **if** *force-cover(v)* **then**
    - **continue**
  - **switch** *expand(v,1)* **do**
    - **case** *Fail: f* **do**
      - **return** *Counter strategy f*
    - **case** *Success* **do**
      - **continue**
- **return** *simulation strategy $\mathcal{L}$*

Theorem 4.1. *Algorithm 1 is sound. For any contextual simulation, if* Strategy-synthesis*($\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$) terminates with a simulation strategy, then $\models \{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$. If* Strategy-synthesis *instead terminates with a simulation counter-strategy then $\not\models \{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$.*

See proof on page 36.

4.1 Expansion

To expand a node n commands, Algorithm 2 constructs the finite horizon game, $\mathcal{G}(\mathcal{L}, v, Q, n)$, which is played as $\mathcal{G}(\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\})$ except that:

- legal plays begin with any place $m \in Places(v)$
- rather than having infinite duration, plays are sequences of moves containing n Verifier places (and at most n Falsifier place), excluding the first move of the play

The first condition ensures that play starts from a place associated with v. The second ensures that any legal play consists of moves corresponding to a sequence of n commands. Every source program command adds a single Verifier place to the play, a target command either adds a single Verifier place or a Verifier place and Falsifier place when the next command is a source command. We exclude the first place in the count as it doesn't correspond to executing some command.

Falsifier wins the play if either of the first two winning conditions from simulation games apply—Verifier makes the first illegal move or Verifier has violated the final conditions of the game. Otherwise Verifier wins the play.

Algorithm 2 computes a winning strategy of the finite-horizon game for the winning player. If Falsifier wins the finite-horizon game, then the algorithm backtracks to v's parent and expands the game with a horizon of $n+1$ (or if v is the root, it returns Falsifier's strategy). If Verifier wins the finite-horizon game, then we may compute a well-labeled unwinding $\mathcal{L}_n$ for it. We may then "paste" $\mathcal{L}_n$ onto v by deleting the sub-tree rooted at v (including any possible covering edges) and then copying $\mathcal{L}_n$ below it.

Algorithm 2: Expand a vertex n commands

```
Procedure expand(v, n)
    switch SimSat(VW(v, n)) do
        case Sat: strat do
            update-tree(v, n, strat);
            return Success
        case Unsat: strat do
            if v = r then
                return Fail: strat
            Let u be v's parent;
            return expand(u, n + 1)

Procedure relabel(v, R)
    Φ(v) ← ¬R(Q_v);
    if Φ(v) ⊨ false then
        delete(v)
    foreach ⟨v, u⟩ ∈ E do
        if v ∈ V then
            G(v, u) ← ¬R(Q_{v,u})
        relabel(u, R)
    foreach u ▷ v s.t. Φ(u) ⊭ Φ(v) do
        ▷ ← ▷ \ {⟨u, v⟩}

Procedure VW(v, n)
    if v ∈ F then
        ψ ← VW_F(S(v), T(v), n)
    else
        b ← F-pred*_e(v);
        ψ ← VW_V(S(v), T(v), b, n)
    return univ-closure(Φ(v) ⇒ ψ)

Function VW_F(l_s, l_t, n)
    if l_s = out_src or n = 0 then
        return true
    Ψ ← true;
    foreach l_s --a--> l'_s do
        if a observable then
            ψ ← VW_V(l'_s, l_t, a, n − 1);
        else
            ψ ← VW_V(l'_s, l_t, None, n − 1);
            ψ ← wp(a, ψ)
        Ψ ← Ψ ∧ ψ
    return Ψ

Procedure update-tree(v, n, strat)
    delete(v);
    copy-strat(v, n, strat);
    Let R be a solution to Rules-of(v);
    relabel(v, R)

Function VW_V(l_s, l_t, b, n)
    if l_t = out_tgt and b ≠ None then
        return false
    if l_s = out_src and l_t = out_tgt then
        return Q
    if n = 0 then return true ;
    Ψ ← false;
    if b = None and l_s ≠ out_src then
        Ψ ← VW_F(l_s, l_t, n)
    foreach l_t --a--> l'_t do
        if a observable then
            ψ ← VW_V(l_s, l'_t, None, n − 1);
            ψ ← match(a, b, ψ);
        else
            ψ ← VW_V(l_s, l'_t, b, n − 1);
            ψ ← pre(a, ψ);
        Ψ ← Ψ ∨ pre(a, ψ)
    return Ψ
```

Finite-Horizon Games. This section describes how to compute well-labeled unwindings for finite-horizon games. We use Figure 4 as a running example, which depicts the processes of expanding node 0 of Figure 2 four commands.

The first step is to encode the game into a quantified LIA formula. In the encoding, Falsifier is the demonic/UNSAT player—controlling conjunctions and universal quantifiers—and Verifier is the angelic/SAT player—controlling disjunctions and existential quantifiers. The finite game is constructed by unrolling the CFG of *src* and *tgt* for n commands starting from $S(v)$ and $T(v)$ and encoding the resulting tree into a LIA formula. Figure 4 (A) shows this unrolling starting from node 0 of Figure 2. Square nodes are where the outgoing commands are from *src*, circles for commands from *tgt*, and half each for nodes where the unrolling has commands from both programs. Figure 4 (B) shows the LIA formula corresponding to this unrolling. We jointly construct the unrolling and the corresponding LIA formula using **VW**, which is split into two mutually recursive functions $\mathbf{VW}_F$ and $\mathbf{VW}_V$. The F variant computes a formula encoding the existence of a non-losing strategy for Verifier for the next n commands, where the next command is from *src* (played by Falsifier), while the V variant encodes when the next command is from *tgt* (played by Verifier). In both variants, l_s represents the control location of *src* and l_t the control location of *tgt*. These control locations dictate which transitions can be played by their respective players (i.e. only transitions corresponding to the commands available at the given control location). The V variant has an additional parameter b, which indicates the communication command that must be matched by Verifier (or None if Falsifier last played a silent command).

The **VW** procedures make use of some auxiliary functions, which we define here. $F\text{-}pred^*_e(v)$ is equal to $F\text{-}pred_e(v)$ if $F\text{-}pred_e(v)$ is observable and unmatched (no V-edge along the path from $F\text{-}pred(v)$ to v is labeled with an observable command); otherwise it is None. The functions **wp** and **pre** denote weakest precondition and preimage predicate transformers, respectively:

$$\begin{aligned}
\mathbf{wp}(x := e, \psi) &\triangleq \psi[x \mapsto e] & \mathbf{pre}(x := e, \psi) &\triangleq \psi[x \mapsto e]\\
\mathbf{wp}(\mathsf{havoc}\ x.\ b, \psi) &\triangleq \forall k.\ b[x \mapsto k] \Rightarrow \psi[x \mapsto k] & \mathbf{pre}(\mathsf{havoc}\ x.\ b,, \psi) &\triangleq \exists k.\ b[x \mapsto k] \wedge \psi[x \mapsto k]\\
\mathbf{wp}([b], \psi) &\triangleq b \Rightarrow \psi & \mathbf{pre}([b], \psi) &\triangleq b \wedge \psi
\end{aligned}$$

For any silent command c and formula ψ, $\mathbf{wp}(c, \psi)$ is a formula satisfied by exactly those valuations that *all* c-successors satisfy ψ, while $\mathbf{pre}(c, \psi)$ is satisfied by exactly those valuations such that *some* c-successor satisfies ψ. $\mathbf{VW}_V$ and $\mathbf{VW}_F$ use **pre** and **wp** to encode the angelic interpretation of the target program and demonic interpretation of the source program. Finally, **match** encodes matching logic for observable commands. If Verifier wants to play send x chan(0) to match send $y + 1$ chan(z), then Verifier must prove that x is equal to $y + 1$ and z is equal to 0. This is the logic that **match** captures. Specifically, **match** takes three parameters (a, b, ψ), where a is an observable command (send/receive) of the target program, b is either an observable command of the source program or None, and ψ is a formula. It computes a formula that captures those valuations under which a and b match, and upon execution result in a valuation that satisfies ψ:

$$\mathbf{match}(a, b, \psi) = \begin{cases} (m = m' \wedge c = c' \wedge \psi) & \text{if } \begin{array}{l} a = \mathsf{send}\ m\ \mathsf{chan}(c),\\ b = \mathsf{send}\ m'\ \mathsf{chan}(c')) \end{array}\\ (c = c' \wedge \forall k.\ \psi[x \mapsto k, y \mapsto k]) & \text{if } \begin{array}{l} a = \mathsf{receive}\ x\ \mathsf{chan}(c),\\ b = \mathsf{receive}\ y\ \mathsf{chan}(c')) \end{array}\\ \mathit{false} & \text{otherwise} \end{cases}$$

After constructing the winning formula $\mathbf{VW}(v, n)$, it is passed to the SimSat algorithm from [Farzan and Kincaid 2016, 2017], which synthesizes a winning strategy for either the SAT player (Verifier) or the UNSAT player (Falsifier). Assuming that $\mathbf{VW}(v, n)$ is satisfiable (if $\mathbf{VW}(v, n)$ is unsatisfiable, the expansion algorithm backtracks), then SimSat produces a SAT strategy. For our purposes, we may think of the SAT strategy as a strategy game unwinding that is equipped with a partial map K that provides terms for the havoc commands in the target program (i.e., witnesses for the existential quantifiers in the winning formula). Figure 4 (C) shows the returned SAT strategy

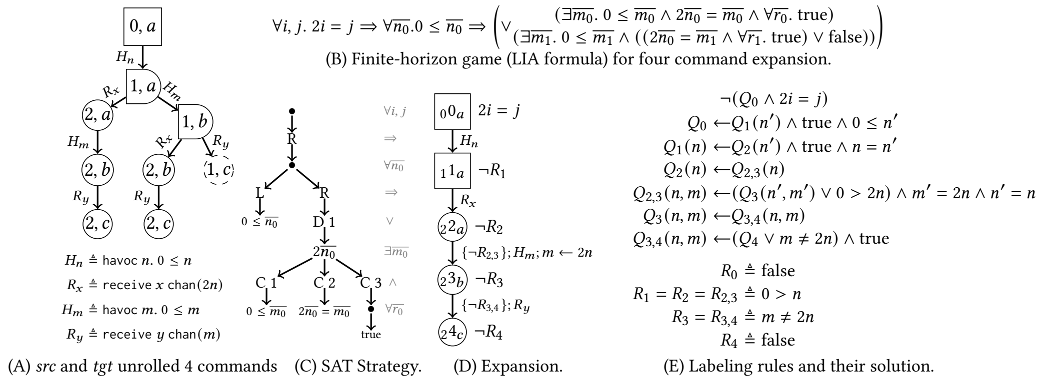


Fig. 4. 4-command expansion of node 0 of Figure 2

for the finite-horizon game in Figure 4 (B). To find suitable labels for Φ and G, we construct and solve a system of constrained horn clauses, *Rules*-of(v). Any solution to these rules provides a valid labeling for Φ and G to ensure that the unwinding is well-labeled. Figure 4 (E) shows the set of rules and their solution needed to construct Φ and G for the expansion in Figure 4 (D).

For a vertex v, define $R(v)$ to be the set of vertices of $\mathcal{L}$ reachable from v using the edges in E. We construct *Rules*-of(v) as follows. For every vertex $u \in R(v)$, we allocate a relation symbol Q_u and for every edge $\langle u, w \rangle$ starting at a V-node $u \in R(v) \cap V$, we allocate a relation symbol $Q_{u,w}$. Q_u represents a set of valuations from which Verifier's strategy loses (starting at u), and similarly $Q_{u,w}$ represents a set of valuations from which Verifier's strategy loses (after taking the edge $\langle u, w \rangle$). To retrieve a labeling from a solution to *Rules*-of(v), we set $\Phi(u)$ to be the negation of the model of Q_u, and $G(u, w)$ to be the negation of the model of $Q_{u,w}$. *Rules*-of(v) are obtained from the contrapositive of the well-labeledness conditions for simulation unwindings (Definition 3.5). For each vertex $u \in R(v)$,

If $u \in F$ then for each $\langle u, v_i \rangle \in E$ add the rule (Consecution)
$$Q_u(X) \leftarrow Q_{v_i}(X') \wedge guard(u, v_i)(X) \wedge act(X, X')$$

If $u \in V$ add the rule (Adequacy)
$$Q_u(X) \leftarrow \bigwedge_{\langle u, v_i \rangle \in E} Q_{u,v_i}(X)$$

and for each $\langle u, v_i \rangle \in E$ add the rule (Consecution)
$$Q_{u,v_i}(X) \leftarrow \big(Q_{v_i}(X') \vee \neg legal(u, v_i)(X)\big) \wedge act(u, v_i)(X, X')$$

If u is final add the rule (Final)
$$Q_n(X) \leftarrow \neg \mathcal{Q}(X)$$

For any $w \in R(v)$ covered by u ($w \triangleright u$) add the rule (Covering)[1]
$$Q_w(X) \leftarrow Q_u(X)$$

If v is the root of the sub-tree then add the rule (Initial)
$$\neg(Q_v(X) \wedge \Phi(v)(X))$$

Intuitively, the local rule for an F-node says Verifier loses the strategy rooted at that node, if for any outgoing edge Falsifier plays according to the command labeling the edge and Verifier loses from the child's sub-tree. For a V-node, the rule says that Verifier loses from that node, if Verifier loses for every outgoing edge. The rules for V-edges say that Verifier loses that edge if the command is infeasible or if Verifier loses the child's subtree.

Given a freshly expanded sub-tree rooted at vertex v, we can be sure that the system CHCs *Rules*-of(v) is non-recursive, and since it is constructed from a winning strategy for Verifier for the finite-horizon game, it must be satisfiable. We may use the model to label the sub-tree (see **relabel** in Algorithm 2) and return success to Algorithm 1.

[1]Expansion does not require handling covering edges, but we consider them here so that we may re-use *Rules*-of in *forced covering*, which does.

Algorithm 3: Attempt to cover a vertex with an already expanded vertex

Procedure force-cover(v)
  $U \leftarrow$ **if** $v \in F$ **then** F **else** V;
  foreach $u \in U \cap X$ *s.t.* $S(u) = S(v)$ *and* $T(u) = T(v)$ **do**
    if $u \in V$ *and* $F\text{-}pred^*_e(u) \neq F\text{-}pred^*_e(v)$ **then**
      continue
    $rules \leftarrow Rules\text{-}of(r) \cup \{Q(v)(X) \leftarrow Q(u)(X)\}$;
    if *rules has some solution* R **then**
      relabel(r, R);
      if $v \in N$ **then**
        if *there exists measures* m'_F *for* $\mathcal{L}_F$ *and* m'_V *for* $\mathcal{L}_V$ **then**
          $m_F \leftarrow m'_F$;
          $m_V \leftarrow m'_V$;
          $\triangleright \leftarrow \triangleright \cup \{\langle v, u\rangle\}$;
          return *true*
      else
        return *true*
  return *false*

4.2 Covering

Forced covering, Algorithm 3, is inspired by McMillan [2006]'s strategy for synthesizing loop invariants in lazy abstraction with interpolants. It searches for any nodes controlled by the same player at the same location as the current node, v. If v is a V-node, we need to ensure that any candidate node u is trying to match the same command as v. If any candidate node u is found, the algorithm constructs a set of (possibly) recursive CHC rules—the same set of rules described for labeling fresh expansions—that will try to relabel the entire tree to ensure that the label of v implies the label of u (without uncovering any previously covered nodes). If a solution is found to this set of rules, the tree is relabeled. Either v was removed from the unwinding (it was annotated with false) or its label implies the label of u (now meeting the covering condition). We finish by trying to compute new measures m_F and m_V such that if the new covering edge is added then both measures will decrease on all corresponding edges—m_F must decrease on every (non-covering) edge out-going from an F node and similarly for m_V and edges out-going from V nodes. If we are successful, we update our measures and add the covering edge and are done. If v was removed we are also successful, and return true. Otherwise, we try the next candidate node or return false if no such candidate exists and Algorithm 1 will attempt to expand v instead.

5 CASE STUDIES

To illustrate the effectiveness of our simulation strategy synthesis approach, we implement our technique in a tool, SimVer. We apply SimVer to a variety of programs and distributed protocols. All experiments were conducted on a desktop running Ubuntu 18.04 LTS equipped with a 4 core Intel(R) Xeon(R) processor at 3.2GHz and 12GB of memory. Each experiment was allotted a maximum of half an hour.

We developed a suite of benchmarks using a simple programming language with deterministic and non-deterministic assignment (`havoc`), `send` and `receive` statements, if statements, while loops, sequential and parallel composition, and parametric parallel composition (`parfor`). We assume that processes may only communicate through sending and receiving messages along

Benchmark	Winner	Size	Base, Abs		Both, Abs		Base, Z3		Both, Z3	
			time	size	time	size	time	size	time	size
Choiceloop1	Verifier	11, 8	4.24s	50	0.60s	23	MO	–	TO	–
Choiceloop2	Verifier	8, 11	2.43s	40	0.90s	30	MO	–	TO	–
Fibloop1	Verifier	13, 13	2.27s	39	0.85s	29	TO	–	TO	–
Fibloop2	Verifier	13, 13	2.24s	38	2.04s	45	MO	–	TO	–
Fibloop3	Verifier	14, 14	7.52s	61	2.23s	43	TO	–	TO	–
Fibloop4	Verifier	14, 14	7.42s	61	1.82s	35	TO	–	TO	–
EvenOdd1	Falsifier	67, 156	MO	–	36.98s	–	MO	–	5.04s	–
EvenOdd2	Falsifier	156, 67	TO	–	1.82s	–	MO	–	2.48s	–

Table 1. Parfor-free benchmarks. We show the winner of the game, the size of the src program and tgt program, and the runtime of SimVer and produced simulation strategy size.

shared channels (i.e., there is no shared memory). Except for `parfor` (discussed below), programs in this language can be simply translated to a control flow graph.

SimVer uses two partial order reduction (POR) techniques to improve upon Algorithm 1. The first POR technique is essentially standard: it reduces the size of the CFG produced for each program when taking the parallel composition of two processes. Processes may only communicate via `send` and `receive`, thus we only consider paths of the product CFG that are unique with respect to observability (i.e. we may re-order sequences of unobservable commands).

The second POR technique reduces the set of unwindings we need by only considering a class of *Lazy* strategies, in which Verifier passes its turn whenever Falsifier plays a silent action (except when Falsifier's vertex belongs to a designated *cutset*). The cutset is a set of locations such that removing them from the CFG results in an acyclic graph. Although Verifier may always legally pass its turn in response to a silent action (even one emanating from the cutset), by allowing the possibility of a non-trivial Verifier response to a silent action in the cutset we may synthesize strategies that take advantage of equi-terminating unobservable loops between programs. We provide details for both POR techniques in Appendix B.2.

We handle programs with a parametric number of processes (includes `parfor` statements), by treating `parfor` statements as an "observable" command (analogously to `send` and `receive`). A `parfor` command is only matched by another `parfor` command. When a source parfor is matched with a target parfor, two verification conditions are induced: (1) both parfors must launch an equal number of processes and (2) the source parfor's body must be simulated by the target parfor's body. The second condition is solved by computing a complete and well-labeled unwinding for the new simulation problem. We further formalize `parfor` statments and the resulting modifications to strategy synthesis in order to handle parfors in Appendix B.1.

Our experimental evaluation aims to answer the following:

(1) Can SimVer prove simulation of real distributed systems?
(2) How do the POR techniques affect performance?
(3) How does the underlying CHC solver affect performance?

We developed a suite of benchmarks that is broken into three categories: simple parfor-free programs, token passing systems a model of distributed processes, and distributed protocols. SimVer is parameterized on three settings: (1) whether or not to use the CFG POR, (2) whether or not to use Lazy strategies, and (3) which underlying CHC solver to use. The two available CHC solvers are Z3's solver [Komuravelli et al. 2016] and a CHC solver based on the polyhedral abstract domain (ABS) using Apron [Jeannet and Miné 2009].

Benchmark	Winner	Base,Abs		CFG,Abs		Turn,Abs		Both,Abs		Base,Z3		CFG,Z3		Turn,Z3		Both,Z3	
		time	size	time	size	time	size	time	size	time	size	time	size	time	size	time	size
General, Ring	Verifier	MO	–	289.7s	280	300.48s	503	13.62s	113	MO	–	MO	–	TO	–	MO	–
General*, Ring	Falsifier	MO	–	MO	–	MO	–	3.0s	–	MO	–	MO	–	MO	–	33.94s	–
General, Lock	Verifier	MO	–	27.11s	131	8.49s	100	5.53s	93	694.15s	157	267.58s	91	288.77s	118	84.2s	73
General*, Lock	Falsifier	MO	–	MO	–	106.03s	–	3.42s	–	MO	–	MO	–	112.98s	–	9.14s	–
General, Ring Lock	Verifier	MO	–	27.85s	134	8.47s	100	7.28s	107	1025.93s	171	429.64s	105	354.32s	128	145.14s	83
General*, Ring Lock	Falsifier	MO	–	MO	–	293.89s	–	7.2s	–	MO	–	MO	–	361.09s	–	17.84s	–
Ring, Ring Lock	Verifier	MO	–	28.23s	135	8.45s	100	7.0s	107	1033.02s	187	540.28s	121	354.76s	131	118.36s	86
Lock, Ring Lock	Verifier	23.72s	130	15.86s	101	11.32s	124	4.1s	89	608.86s	134	245.91s	91	267.6s	116	77.31s	74

Table 2. Token Passing benchmarks. We show the winner of the game, runtime of SimVer, and produced simulation strategy size. * Denotes the faulty version.

We run SimVer in each of the eight configurations on each benchmark. Each benchmark is run 10 times. Tables 1, 2, and 3 report the mean runtime or failure status of each experiment (either timed out or ran out of memory).

5.1 Token Passing Systems

Token passing systems are a formalism for modeling distributed protocols [Chandy and Lamport 1985]. Here we represent four similar token passing systems: General, Ring, Lock, and Ring Lock. All four programs have N nodes run in parallel that acquire and release some number of tokens. Each is represented by a non-deterministic choice for the number of processes (and tokens for General and Ring). The parent process spawns two children processes, one will create some number of tokens and close, while the other sub-process will execute a parfor running the N nodes of the token passing system.

General: The General token system has M tokens, may be acquired and released by any node (any node may acquire up to all of the tokens). When a node releases a token, it does so by sending it to another node. When sending, a node may send to any of its neighbors (including itself). In the buggy version of General, when releasing a token, a node must send it to a neighbor that is not itself. This violates simulation when there is only 1 node in the system. Both the faulty and non-faulty variant contain 126 CFG nodes.

Ring: The Ring token systems also has M tokens, but the nodes form a ring topology. When a node releases a token, it must send the token to the next node in the ring. This system is simulated by General but not any of the other systems. The Ring token passing system contains 150 CFG nodes.

Lock: The Lock token system has a single token, it is designed to operate as a lock. Similar to General, the Lock system is allowed to send its token to any of the other nodes within the system. Like Ring, this system is simulated by General but none of the other systems. The Lock token passing system contains 58 CFG nodes.

Ring Lock: The Ring Lock token system has a single token, and the nodes form a ring topology. This protocol is simulated by all others (not including the faulty General system). The Ring Lock token passing system contains 70 CFG nodes.

5.2 Distributed Protocols

The final set of benchmarks contains several varieties of replicated state machine algorithms and a leader election protocol. For each, we implement two variants: *abstract* models all degrees of freedom in the protocol (i.e. when a choice of implementation is allowed, we use havoc expressions to abstract away the choice), and *concrete* selects a particular choice for each implementation decision. For each protocol, the desired goal is to show that *abstract* weakly simulates *concrete*.

Replicated State Machines: Two Phase Commit ([Lampson and Sturgis 1979] and [Gray 1978]), Two Phase Commit with Apportioned Queries (2PAQ [Mohan and Murphy 2017]), Chain Replication ([Van Renesse and Schneider 2004]), Chain Replication with Apportioned Queries (CRAQ [Terrace

Benchmark	Winner	Base, ABS		CFG, ABS		Turn, ABS		Both, ABS		Base, Z3		CFG, Z3	
		time	size	time	size	time	size	time	size	time	size	time	size
Two Phase Commit (kv)	Verifier	448.11s	403	74.54s	113	259.81s	400	26.6s	89	MO	–	TO	–
Two Phase Commit (fib)	Verifier	MO	–	MO	–	294.58s	334	30.65s	169	MO	–	MO	–
2PAQ (kv)	Verifier	TO	657	592.7s	371	MO	–	132.24s	258	MO	–	MO	–
2PAQ (fib)	Verifier	TO	–	MO	–	185.27s	339	791.65s	743	MO	–	MO	–
Chain Replication (kv)	Verifier	25.8s	128	26.11s	128	6.77s	94	6.71s	94	427.18s	125	426.07s	125
Chain Replication (fib)	Verifier	32.24s	142	33.81s	142	12.06s	110	11.13	110	MO	–	MO	–
CRAQ (kv)	Verifier	29.59s	120	29.7s	120	6.35s	91	6.44s	91	MO	–	535.88s	117
CRAQ (fib)	Verifier	35.75s	132	36.83s	132	53.89s	185	49.96s	185	MO	–	MO	–
Parallel Backup	Verifier	TO	–	TO	–	TO	–	1255.46s	481	TO	–	MO	–
Sequential Backup	–	TO	–	TO	–	TO	–	TO	–	TO	–	MO	–
Toy Leader Election	–	TO	–	TO	–	MO	–	MO	–	MO	–	TO	–
Leader Election (delay)	–	MO	–	TO	–	TO	–	TO	–	TO	–	TO	–

Table 3. Distributed protocol benchmarks. We show the winner, runtime of SimVer, and simulation strategy size.

and Freedman 2009]), and Parallel and Sequential Primary Backup ([Budhiraja et al. 1993]) are all forms of replicated state machine protocols. Each of these protocols have a designated leader, and N followers. The goal of these protocols is to have the state of the Leader be replicated on each of the followers. The primary differences between these protocols is either in the failure model or topology of the system. Two Phase Commit, 2PAQ, Sequential Backup, and Primary backup have a flat topology—there is one leader that has N children. Two Phase Commit and 2PAQ first have the leader node stage (prepare) any state update, ask each child if they could successfully stage the write, then commit the write if each child could perform the write, otherwise the write is aborted. In two phase commit, only the leader is able to handle read requests. In 2PAQ, any node may handle read requests. Both Sequential and Parallel backup relax the failure model of Two phase commit and 2PAQ, and are able to replicate with fewer messages between the leader and its followers. Parallel backup propagates writes to all children then receives acknowledgments from all of its children, while sequential backup handles each child in turn. Chain replication and CRAQ are formed in a linked list (or chain) topology. One end of the Linked list is the Head and handles every write request. The other end is the tail and handles write requests. When a write occurs, the head will propagate the write request down the chain until it reaches the tail. The tail will then either accept or reject the write and the result is propagated back up the chain to the head. CRAQ is the same protocol, except that any node in the system may respond to read requests.

For Two Phase Commit, 2PAQ, Chain Replication, and CRAQ we implement two concrete variants. The first variant implements a single-key key-value store on top of the protocol as our *concrete* system. The second variant is similar, but rather than simply reading and writing to the key, the second variant will compute the nth Fibonacci number and add that to the current value of the key. For parallel and sequential backup, the *concrete* system is a backup system with only one backup.

Leader Election: The Leader Election protocol chooses the leader by finding the node with the largest id within the system [Chang and Roberts 1979]. The protocol consists of N nodes in a ring topology, where each node is given an unique id. The protocol begins by having each node pass their id to their right neighbor. If a node receives an id larger than their own, then the node continues passing the id to the next node. Once a node receives its own id then it becomes the leader and may make some number of decisions to send to all of their neighbors. In the base protocol the number of decisions and the choice of value to send are both non-deterministic. We implement two other variants. Both perform the election process but only make a single decision. The second variant adds a random delay before every send command.

5.3 Performance

In Tables 1, 2, and 3, we summarize the performance of SimVer. In Table 1 we drop configurations using only one of the PORs. The CFG POR only applies to programs with multiple processes—Choiceloop and Fibloop are sequential. The EvenOdd benchmarks fail when using only one of the two POR. In Table 3 we drop the Z3 columns that uses Lazy strategies as these SimVer configurations failed to solve any of the benchmarks.

In all experiments that do not fail (except "Craq (fib)"), we find that the Lazy strategy POR significantly improves runtime performance and reduces the size of the produced unwinding. In the "Craq (fib)" benchmark, forcing the use of a Lazy strategy resulted in poor alignment of the abstract and concrete variants that resulted in requiring a slightly larger strategy. We find the use of Lazy strategies especially important for benchmarks that Falsifier wins—Lazy strategies can result in exponentially smaller formulas during back-tracking.

In Table 2, we see that with neither reduction, only one benchmark was able to be solved within half an hour. With only one of the two reductions, most were solvable in under five minutes. However with both, SimVer was able to solve all of the token passing benchmarks in under 15 seconds.

In Table 3, we see that our implementation had a harder time finding simulation between the protocols and their implementations. Using both reductions and the "Abs" CHC solver, SimVer was able to solve all but three of the protocol benchmarks. As expected, we see SimVer performs better on the "kv" benchmarks as compared to the "fib"—there is a larger gap between the protocol and "fib" variant due to the loops used to compute Fibonacci numbers. Perhaps the hardest benchmarks are the parallel and sequential backup—as the benchmarks effectively require proving the two protocols are equivalent when there is only one backup. Examining SimVer's performance on the leader election benchmarks revealed that SimVer made a poor initial choice of strategy that resulted in a lot of back-tracking later on.

In all the benchmarks, we find that the "ABS" SimVer configurations had better run-time performance than the "Z3" configurations, while the "Z3" configurations resulted in smaller strategies. The Z3 CHC solver tended to find more generalizable loop invariants, which often came at the cost of run-time performance or even with Z3 diverging due to the complex forced-covering rules. The "Abs" solver, was quick but produced less generalizable invariants or failed to find loop invariants during forced-covering causing the algorithm to continue expanding the node. Striking a balance between finding generalizable loop invariants and reasonable run-time performance is important. As SimVer is sensitive to the underlying CHC solver, improvements in CHC solving will correspondingly help SimVer during forced covering.

Our experiments show that simulation strategy synthesis can be used to prove and refute simulation between non-deterministic infinite state programs. There are only three benchmarks that were unsolved by all SimVer configurations. The use of both reductions is crucial in helping SimVer scale to more benchmarks. We note that SimVer performs best when the reason for a strategy's correctness is relatively "local."

6 RELATED WORK

Computing Simulations. There is a long line of literature on computing simulations for finite systems [Abdulla et al. 2008; Baier et al. 2004; Bulychev et al. 2007; Dovier et al. 2001; Etessami et al. 2005; Groote and Vaandrager 1990; Li 2009; Paige and Tarjan 1987], motivated primarily by the use of simulation (and bisimulation) relations as a state-space reduction technique in model checking [Escobar and Meseguer 2007; Fisler and Vardi 1999]. Techniques have also been developed for proving that an infinite-state system simulates a finite-state system [Chaki et al. 2004; Chutinan and Krogh 2001; Henzinger et al. 1995; Jančar et al. 2001; Jonsson and Parrow 1993]. The major point of contrast between our algorithm and this body of work is that SimVer is designed to prove (or refute) simulation between systems that are *both* infinite-state. Perhaps most similar to our work is Chaki et al. [2004], which gives an algorithm for proving that a finite-state protocol simulates an infinite-state system following the counter-example guided abstraction refinement (CEGAR) paradigm. For the particular case that the target program is finite-state, the difference between Chaki et al. [2004]'s method and ours is analogous to the difference between CEGAR [Clarke et al. 2000] and lazy abstraction with interpolants [McMillan 2006]: instead of finding a finite global abstraction that admits a winning strategy, we iteratively expand a partial strategy until it is complete.

Relational Logics. Relational logics (such as Relational Hoare Logic [Benton 2004]) are program logics that, like contextual simulations, relate the behaviors of two or more programs together [Barthe et al. 2009; Gäher et al. 2022; Godlin and Strichman 2008; Hur et al. 2014; Lucanu and Rusu 2015; Song et al. 2023; Yang 2007]. There has been a great deal of work on automated verification of relational properties. A prominent class of techniques use *product programs* to reduce relational verification to classical verification by combining two (or more) programs into one [Barthe et al. 2016; Churchill et al. 2019; Sharma et al. 2013].

The closest relational logics to contextual simulation are those in Benton [2004], Lucanu and Rusu [2015], Hur et al. [2014], [Song et al. 2023], and [Gäher et al. 2022]. For unobservable, straight-line, deterministic programs, contextual simulation can be seen as equivalent to proving both Benton [2004]'s relational Hoare logic judgment and relative termination of the two programs. While Lucanu and Rusu [2015] can automatically prove observational equivalence of two programs, the technique requires the user to define when two CFG locations are observably equivalent. This is analogous to checking whether a given relation is a simulation rather than synthesizing a simulation. Hur et al. [2014] and [Gäher et al. 2022] both prove observational equivalence (stuttering bisimulation) of ML-like programs in Coq. The program models consider an alphabet with a single observable action (a reduction step) rather than considering programs that communicate with some outside environment. [Song et al. 2023] is similar to [Hur et al. 2014] and [Gäher et al. 2022] but uses trace refinement rather than simulation. In light of methods for automating verification based on product programs, one may view SimVer as an on-the-fly construction of a product program, interpreting one program with demonic non-determinism and the other with angelic non-determinism (rather than both demonic).

Infinite-state games. Our method for proving contextual simulations is based on reducing the problem to solving a class of infinite-state games of infinite duration. Methods for solving such games include [Ball and Kupferman 2006; Beyene et al. 2014; De Alfaro et al. 2001; Farzan and Kincaid 2017]. Our method is closely related to Farzan and Kincaid [2017]'s technique for solving reachability games. The largest difference between SimVer and [Farzan and Kincaid 2017] is that Farzan and Kincaid [2017]'s reachability games require the two players to strictly alternate turns, while in weak simulation games, Verifier may take arbitrarily many steps to match the move of Falsifier. Moreover, SimVer exploits some additional structure that is present in weak simulation games, including the graph structure of programs (e.g., in the covering algorithm) and the POR techniques described in Section 5.

REFERENCES

Parosh A Abdulla, Ahmed Bouajjani, Lukáš Holík, Lisa Kaati, and Tomáš Vojnar. 2008. Computing simulations over tree automata. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 93–108.

Christel Baier, Holger Hermanns, and Joost-Pieter Katoen. 2004. Probabilistic weak simulation is decidable in polynomial time. *Information processing letters* 89, 3 (2004), 123–130.

Thomas Ball and Orna Kupferman. 2006. An abstraction-refinement framework for multi-agent systems. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. IEEE, 379–388.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2016. Product programs and relational program logics. *Journal of Logical and Algebraic Methods in Programming* 85, 5, Part 2 (2016), 847–859.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 2009. Formal certification of code-based cryptographic proofs. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 90–101.

Saddek Bensalem, Ahmed Bouajjani, Claire Loiseaux, and Joseph Sifakis. 1992. Property preserving simulations. In *International Conference on Computer Aided Verification*. Springer, 260–273.

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *ACM SIGPLAN Notices*, Vol. 39. ACM, 14–25.

Tewodros Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. 2014. A constraint-based approach to solving games on infinite graphs. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 221–233.

Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. 1993. The primary-backup approach. *Distributed systems* 2 (1993), 199–216.

PE Bulychev, IV Konnov, and VA Zakharov. 2007. Computing (bi) simulation relations preserving CTL* X. for ordinary and fair Kripke structures. *Mathematical Methods and Algorithms* 12 (2007).

Sagar Chaki, Edmund M Clarke, Alex Groce, Somesh Jha, and Helmut Veith. 2004. Modular verification of software components in C. *IEEE Transactions on Software Engineering* 30, 6 (2004), 388–402.

K Mani Chandy and Leslie Lamport. 1985. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)* 3, 1 (1985), 63–75.

Ernest Chang and Rosemary Roberts. 1979. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Commun. ACM* 22, 5 (1979), 281–283.

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic Program Alignment for Equivalence Checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 1027–1040. https://doi.org/10.1145/3314221.3314596

Alongkrit Chutinan and Bruce H Krogh. 2001. Verification of infinite-state dynamic systems using approximate quotient transition systems. *IEEE Transactions on automatic control* 46, 9 (2001), 1401–1410.

Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *Computer Aided Verification*, E. Allen Emerson and Aravinda Prasad Sistla (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 154–169.

Luca De Alfaro, Thomas A Henzinger, and Rupak Majumdar. 2001. Symbolic algorithms for infinite-state games. In *International Conference on Concurrency Theory*. Springer, 536–550.

Agostino Dovier, Carla Piazza, and Alberto Policriti. 2001. A fast bisimulation algorithm. In *International Conference on Computer Aided Verification*. Springer, 79–90.

Santiago Escobar and José Meseguer. 2007. Symbolic model checking of infinite-state systems using narrowing. In *International Conference on Rewriting Techniques and Applications*. Springer, 153–168.

Kousha Etessami, Thomas Wilke, and Rebecca A Schuller. 2005. Fair simulation relations, parity games, and state space reduction for Büchi automata. *SIAM J. Comput.* 34, 5 (2005), 1159–1175.

Azadeh Farzan and Zachary Kincaid. 2016. Linear Arithmetic Satisfiability via Strategy Improvement.. In *IJCAI*. 735–743.

Azadeh Farzan and Zachary Kincaid. 2017. Strategy synthesis for linear arithmetic games. *Proceedings of the ACM on Programming Languages* 2, POPL (2017), 1–30.

Kathi Fisler and Moshe Y Vardi. 1999. Bisimulation and model checking. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Springer, 338–342.

Lennard Gäher, Michael Sammler, Simon Spies, Ralf Jung, Hoang-Hai Dang, Robbert Krebbers, Jeehoon Kang, and Derek Dreyer. 2022. Simuliris: a separation logic framework for verifying concurrent program optimizations. *Proceedings of the ACM on Programming Languages* 6, POPL (2022), 1–31.

Benny Godlin and Ofer Strichman. 2008. Inference rules for proving the equivalence of recursive procedures. *Acta Informatica* 45, 6 (2008), 403–439.

James N Gray. 1978. Notes on data base operating systems. In *Operating Systems*. Springer, 393–481.

Jan Friso Groote and Frits Vaandrager. 1990. An efficient algorithm for branching bisimulation and stuttering equivalence. In *International Colloquium on Automata, Languages, and Programming*. Springer, 626–638.

Monika Rauch Henzinger, Thomas A Henzinger, and Peter W Kopke. 1995. Computing simulations on finite and infinite graphs. In *Proceedings of IEEE 36th Annual Foundations of Computer Science*. IEEE, 453–462.

Chung-Kil Hur, Georg Neis, Derek Dreyer, and Viktor Vafeiadis. 2014. *A logical step forward in parametric bisimulations.* Technical Report. Citeseer.

Petr Jančar, Antonín Kučera, and Richard Mayr. 2001. Deciding bisimulation-like equivalences with finite-state processes. *Theoretical Computer Science* 258, 1-2 (2001), 409–433.

Bertrand Jeannet and Antoine Miné. 2009. Apron: A library of numerical abstract domains for static analysis. In *International Conference on Computer Aided Verification*. Springer, 661–667.

Bengt Jonsson and Joachim Parrow. 1993. Deciding bisimulation equivalences for a class of non-finite-state programs. *Information and computation* 107, 2 (1993), 272–302.

Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2016. SMT-based model checking for recursive programs. *Formal Methods in System Design* 48, 3 (2016), 175–205.

Butler Lampson and Howard E Sturgis. 1979. Crash recovery in a distributed data storage system. (1979).

Weisong Li. 2009. Algorithms for computing weak bisimulation equivalence. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 241–248.

Dorel Lucanu and Vlad Rusu. 2015. Program equivalence by circular reasoning. *Formal Aspects of Computing* 27, 4 (2015), 701–726.

Kenneth L McMillan. 2006. Lazy abstraction with interpolants. In *International Conference on Computer Aided Verification*. Springer, 123–136.

Robin Milner. 1971. *An algebraic definition of simulation between programs.* Citeseer.

Robin Milner. 1989. *Communication and concurrency*. Vol. 84. Prentice hall New York etc.

Divyarthi Mohan and Charlie Murphy. 2017. Two Phase Commit With Apportioned Queries (2PAQ). https://medium.com/princeton-systems-course/two-phase-commit-with-apportioned-queries-2paq-376701c2a5b4

Rocco de Nicola and Frits Vaandrager. 1990. Action versus state based logics for transition systems. In *LITP Spring School on Theoretical Computer Science*. Springer, 407–419.

Robert Paige and Robert E Tarjan. 1987. Three partition refinement algorithms. *SIAM J. Comput.* 16, 6 (1987), 973–989.

Joachim Parrow, Tjark Weber, Johannes Borgström, and Lars-Henrik Eriksson. 2017. Weak nominal modal logic. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*. Springer, 179–193.

Doron Peled. 1998. Ten years of partial order reduction. In *International Conference on Computer Aided Verification*. Springer, 17–28.

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *OOPSLA*. 391–406.

Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 2023. Conditional Contextual Refinement. *Proceedings of the ACM on Programming Languages* 7, POPL (2023), 1121–1151.

Jeff Terrace and Michael J Freedman. 2009. Object Storage on CRAQ: High-Throughput Chain Replication for Read-Mostly Workloads.. In *USENIX Annual Technical Conference*. San Diego, CA, 1–16.

Rob J Van Glabbeek. 2001. The linear time-branching time spectrum I. The semantics of concrete, sequential processes. In *Handbook of process algebra*. Elsevier, 3–99.

Robbert Van Renesse and Fred B Schneider. 2004. Chain Replication for Supporting High Throughput and Availability.. In *OSDI*, Vol. 4.

Hongseok Yang. 2007. Relational separation logic. *Theoretical Computer Science* 375, 1-3 (2007), 308–334.

$$\frac{}{\texttt{skip} \overset{[true]}{\dashrightarrow} \blacksquare} \qquad \frac{S \text{ is } \texttt{x := e}}{S \overset{S}{\dashrightarrow} \blacksquare} \qquad \frac{S \text{ is } \texttt{havoc x. b}}{S \overset{S}{\dashrightarrow} \blacksquare} \qquad \frac{S \text{ is } \texttt{send e chan(c)}}{S \overset{S}{\dashrightarrow} \blacksquare}$$

$$\frac{S \text{ is } \texttt{receive x chan(c)}}{S \overset{S}{\dashrightarrow} \blacksquare} \qquad \frac{}{\text{if } b \text{ then } S \text{ else } T \overset{[b]}{\dashrightarrow} S} \qquad \frac{}{\text{if } b \text{ then } S \text{ else } T \overset{[\neg b]}{\dashrightarrow} T}$$

$$\frac{}{\text{while } b \text{ do } S \text{ done} \overset{[\neg b]}{\dashrightarrow} \blacksquare} \qquad \frac{}{\text{while } b \text{ do } S \text{ done} \overset{[b]}{\dashrightarrow} S; \text{while } b \text{ do } S \text{ done}}$$

$$\frac{S \overset{a}{\dashrightarrow} S'}{S;T \overset{a}{\dashrightarrow} S';T} \qquad \frac{S \overset{a}{\dashrightarrow} S'}{S \| T \overset{a}{\dashrightarrow} S' \| T} \qquad \frac{T \overset{a}{\dashrightarrow} T'}{S \| T \overset{a}{\dashrightarrow} S \| T'}$$

Fig. 5. Inference rules for program CFG relation

A SYNTACTIC PROGRAMS AND THEIR CFG

In this section we give the syntactic programs we use in Section 5 to express our benchmarks.

A.1 Message Passing Programs

We consider a simple language of multi-threaded message-passing integer programs. The syntax of statements is given below:

$$
\begin{aligned}
\langle \mathit{stmt} \rangle ::= \; & \blacksquare \mid \text{skip} \mid \langle \mathit{var} \rangle := \langle \mathit{exp} \rangle \mid \text{havoc } \langle \mathit{var} \rangle . \, \langle \mathit{bexp} \rangle \\
\mid \; & \text{send } \langle \mathit{exp} \rangle \text{ chan}(\langle \mathit{exp} \rangle) \mid \text{receive } \langle \mathit{var} \rangle \text{ chan}(\langle \mathit{exp} \rangle) \\
\mid \; & \langle \mathit{stmt} \rangle \, ; \langle \mathit{stmt} \rangle \mid \langle \mathit{stmt} \rangle \parallel \langle \mathit{stmt} \rangle \\
\mid \; & \text{if } \langle \mathit{bexp} \rangle \text{ then } \langle \mathit{stmt} \rangle \text{ else } \langle \mathit{stmt} \rangle \\
\mid \; & \text{while } \langle \mathit{bexp} \rangle \text{ do } \langle \mathit{stmt} \rangle \text{ done}
\end{aligned}
$$

We reuse the same language of expressions and Boolean expressions from Section 2. The language of syntactic programs includes any statement that does not include $\blacksquare$, where $\blacksquare$ denotes the halting program. Processes are created with the parallel composition operator $\parallel$ (in Section B.1, we further extend the language with parfor, a parametric n-ary parallel repetition statement). Processes may communicate by passing messages (using `send` and `receive`) along shared channels, which are identified by integers; processes do not share memory. We treat $\blacksquare; S$ as equal to S and $\blacksquare \| \blacksquare$ as equal to $\blacksquare$.

A.2 Control Flow Graphs

To give meaning to syntactic programs, we map each syntactic program p to a control flow graph, $CFG(p)$. To simplify this compilation process, we assume that when two statements are parallelly composed, they write to a disjoint set of variables and that each child process does not write to any variable that a parent process reads or writes. The compilation process from a syntactic program to a CFG loses the notion of processes, these assumptions ensure that the compilation process doesn't inadvertently introduce memory sharing between processes by having clashing local variable names.

We represent the CFG of syntactic programs using a labeled binary relation $\dashrightarrow_{\text{stmt}}$ over statements. Figure 5 displays the rules defining $\dashrightarrow_{\text{stmt}}$. For any syntactic program p, its control flow graph is $CFG(p) = \langle \text{stmt}, \dashrightarrow_{\text{stmt}}, p, \blacksquare \rangle$. The semantics of a syntactic program is given by the semantics assigned to its CFG as defined in Section 2.

$$\frac{[\![e_l \le e_u]\!]_\lambda}{\lambda \triangleright \left(\begin{matrix}\text{parfor } id.\, e_l \le id \le e_u \\ \text{do } S \text{ done}\end{matrix}\right) \xrightarrow{\tau} \lambda \triangleright \left(id := e_l; S \| \begin{matrix}\text{parfor } id.\, e_l{+}1 \le id \le e_u \\ \text{do } S \text{ done}\end{matrix}\right)}$$

$$\frac{[\![e_l \le e_u]\!]_\lambda = \text{false}}{\lambda \triangleright \left(\begin{matrix}\text{parfor } id.\, e_l \le id \le e_u \\ \text{do } S \text{ done}\end{matrix}\right) \xrightarrow{\tau} \lambda \triangleright \blacksquare} \qquad \frac{a = \text{parfor } id.\, e_l \le id \le e_u.\ \text{do } S \text{ done}}{a \overset{a}{\dashrightarrow} \blacksquare}$$

Fig. 6. Additional transition and CFG rules for parfor.

B EXTENSIONS AND ALGORITHMIC IMPROVEMENTS

In this section we describe extensions that enable our method to handle larger and more realistic programs: we enrich the language with a parallel repetition construct `parfor`, and formalize the two partial order reduction strategies (as discussed in Section 5) that reduce the search space for simulation proofs using the algorithm described in Section 4.

B.1 Parfor: n-ary Parallel Repetition

Many interesting programs and distributed systems use a parametric number of processes. To handle this paradigm, we introduce a parametric parallel composition operator, `parfor`. We detail the changes to the programming language, its semantics, and control flow graph. We additionally update our definition of simulation game unwindings and algorithms to handle the extended language.

The (`parfor` $id.$ $e_l \le id \le e_u$ `do S done`) construct runs n copies of its body in parallel, one copy for each thread id in the range $[e_l, e_u]$. We extend the grammar of programs to include `parfor` as follows:

$$\langle stmt \rangle ::= \ \ldots \mid \text{parfor } \langle var \rangle.\ \langle exp \rangle \le \langle var \rangle \le \langle exp \rangle \text{ do } \langle stmt \rangle \text{ done}$$

Unlike the syntactic programs in Section A, `parfor` is difficult to represent as a finite CFG. Specifically, unless the range $[e_l, e_u]$ is statically known, the approach used to represent the other syntactic programs will yield an infinite size CFG. We can provide an operation semantics of parfor based on unrolling the parfor using the binary parallel composition operator analogous to the unrolling semantics of while using sequential composition. The first rule for parfor handles the case where the range is non-empty: it peels off the lowest valued identifier in the range and runs it in parallel with the remainder of the parfor. The second rule transitions to the empty program (final state) when the parfor's range is empty.

If we treated the CFG construction of parfor similar to its operational semantics (by repeated unrolling), the set of reachable statements from a parfor statement would be infinite. As described in Section 5, we instead choose to treat parfor as an "observable" command similar to how we treat `send` and `receive`. We modify the set of commands to include every parfor program. Figure 6 gives the CFG for parfor programs.

We now update the definition of well-labeled unwindings to support `parfor`. We treat parfors as an observable command. While in Figure 6, we see that when a parfor statement unrolls it executes a τ action, if the body of the parfor may execute a send or receive, then the parfor as a whole is observable. The way we handle this is by matching a parfor of the source program with a parfor of the target program, in the same manner as we did for sends and receives. This induces additional verification conditions and requirements on when a source parfor is matched by a target parfor. Note that this rule is incomplete—it is possible *semantically* for a parfor program to be simulated

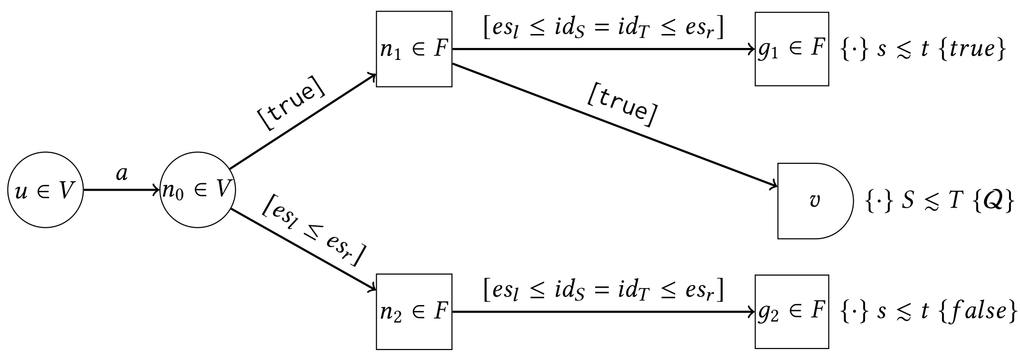


Fig. 7. A parfor gadget to inline the terminating and non-terminating subgames.

by (or to simulate) a parfor free program. However, this strategy is suitable for our goal of proving per-node simulation.

We update *legal* with a new clause $legal(u,v) = (es_l > es_r \wedge et_l > et_r) \vee (es_l = et_l \wedge es_r = et_r)$ when $L(u,v)$ is parfor $id_T.et_l \leq id_T \leq et_r$ do t done and F-$\text{pred}_e(u)$ is parfor $id_S.es_l \leq id_S \leq es_r$ do s done. *act* remains unchanged. This change to *legal* ensures that either both parfors do not execute or they both execute with an equal range of thread ids. The existing well-labeledness constraints remain unchanged; however, we add one more constraint *subgame* to the set of well-labeled constraints.

Subgame: Every V-edge $\langle u, v\rangle$ labeled with parfor $id_T.et_l \leq id_T \leq et_r$ do t done where $F\text{pred}_e(u)$ is parfor $id_S.es_l \leq id_S \leq es_r$ do s done. There is a well-labeled unwinding for the game $\mathcal{G}(\{\Phi(u)\}\ t \lesssim s\ \{true\})$.

A labeled unwinding is complete only if it and every associated subgame's labeled unwinding game is complete. Thus a well-labeled and complete game unwinding must have a well-labeled and complete unwinding associated with every V-edge labeled with a parfor proving simulation between the source and target parfor's bodies.

To satisfy the modified definitions of well-labeledness and complete we must now compute a complete well-labeled unwinding for each induced sub-game. Rather than eagerly (at match time) or lazily (after finding a well-labeled and complete unwinding for the parent game) computing these induced unwindings—by recursively calling Algorithm 1 for each matched parfor—, we inline each subgame using a *parfor gadget*. This allows us to simultaneously compute the strategy for the root game and any induced subgames. Specifically, this enables us to jointly refine the parent and child strategies—if Verifier loses a subgame, the algorithm can (attempt to) improve the parent strategy so that the induced subgame is more favorable. We additionally allow reuse of work between similar induced sub-games—we allow covering a node by another if they share the same sub-game label even if the sub-games are induced by different Verifier edges. Inlining is accomplished as follows. We augment simulation game trees with an additional field *game* that maps each node of $\mathcal{L}$ with its subgame, taking the form $\{\cdot\}\ s \lesssim t\ \{\psi\}$ (note the precondition is omitted, since it is the same as the label of the subgame's root). A node has the same game as its immediate ancestor, with the exception of the nodes introduced by the parfor gadget, which is introduced for each pair of matching parfors.

Figure 7 depicts the gadget used to inline induced subgames. We actually play two subgames for each parfor. One where we try to prove the post-condition false (meaning the matched parfors do not terminate) and the other with the post-condition true (meaning the subgames are allowed to terminate in any state). If we are able to find a strategy for the false post-condition, then we do not need to continue finding a strategy for the remainder of the parent game (as play never

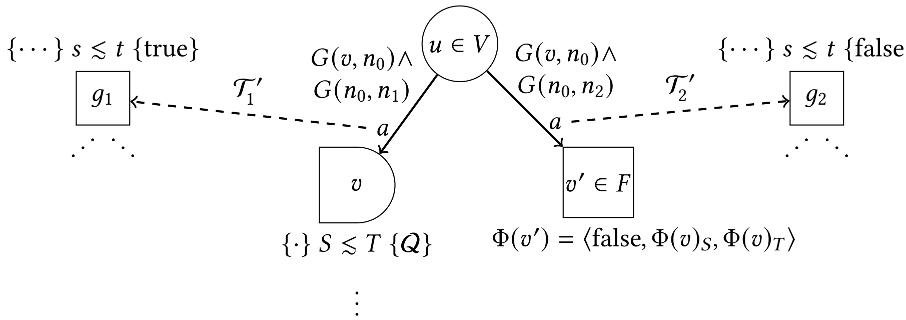


Fig. 8. The well labeled game trees produced by removing a parfor gadget.

returns back to the parent game). The edge from u to n_0 is the edge labeled with a, Verifier's parfor action. Because $n_0 \in V$, Verifier controls which sub-game to play: the terminating game or the non-terminating game (g_1 and g_2 respectively). If Verifier chooses to play the non-terminating game, Verifier must prove that the parfors actually execute. In both the terminating and non-terminating games, we allow Verifier to assume that the ids are equal and are bounded by the given range. If Verifier plays the terminating game, then Verifier must also have a strategy for v, which represents the remainder of the parent game after the parfor is played. Algorithm 1 remains unchanged other than initializing the *game* variable. In Algorithm 2, the only change is the addition of the parfor gadget in **VW** when Verifier plays a matching parfor. In Algorithm 3, we require that nodes are only covered by nodes labeled with the same subgame. Otherwise, the algorithms remain unchanged.

In Section 4, we maintained the invariant that the unwinding $\mathcal{L}$ is always well-labeled. In the updated algorithm to handle parfor, we maintain the invariant that the unwinding $\mathcal{L}$ is well-labeled modulo parfor gadgets. That is, if we remove each parfor gadget, we produce a set of well labeled unwindings—one for the parent game and each introduced sub-game. In Figure 8, we show how the parfor gadget in Figure 7 is transformed into three well-labeled unwindings (the parent game and both sub-game trees). We remove interior nodes n_0, n_1, and n_2. We add an edge from u to v (and a fresh node v') labeled with the parfor action Verifier played. The edge to v represents when Verifier chose to play the terminating game, and v' the non-terminating game. Both are guarded by the parfor edge's original guard ($G(u, n_0)$). Each is additionally guarded with Verifier's choice to play that game (i.e. $G(n_0, n_1)$ and $G(n_0, n_2)$ respectively). The nodes g_1 and g_2 now become the root of their own simulation unwinding for the corresponding game. Thus, we have removed the parfor gadget and shown the existence of the unwindings satisfying the sub-game constraint. When Algorithm 1 terminates with $\mathcal{L}$, then every node of every subgame is either expanded or covered (by a node of the same game). Unlike the original version of the algorithm, if the modified algorithm terminates without finding a simulation strategy, the produced counter-example does not disprove simulation: it only disproves *per-node* simulation.

Theorem B.1. *If the modified version of Algorithm 1 terminates with some unwinding $\mathcal{L}$ then $\{\mathcal{P}\}$ src $\lesssim$ tgt $\{Q\}$ the input contextual simulation is valid.*

Proof Sketch. The produced unwinding is well-labeled and complete. As described above, we maintained the invariant that $\mathcal{L}$ is well-labeled modulo parfor gadgets. Above we gave the transformation from an unwinding containing parfor gadgets to a set of well-labeled unwindings for each sub-game. After terminating each node of every game was either expanded or covered. Thus each unwinding is well-labeled and complete. The proof proceeds by inducting on the depth of sub-games within $\mathcal{L}$. If there are no sub-games then Theorem 4.1 proves the conclusion. By the inductive hypothesis, every contextual simulation labeling matched parfor edges are valid. Let $g'_{\mathcal{L}}$ be the strategy for Verifier as described in Section 3. We construct a new strategy $g_{\mathcal{L}}$. Let s be a position ending in a Verifier place of the game $G(\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$. If Falsifier's most recent action was not within a parfor, then $g_{\mathcal{L}}$ follows the strategy of $g'_{\mathcal{L}}$. If Falsifier's move was to unroll a parfor one more time then $g_{\mathcal{L}}$ will do the same. If Falsifier plays an action from the body of thread i from a parfor. Then Verifier will play a matching action for it's thread i using the strategy for the induced subgame (note this strategy exists and is winning because the IH allows us to assume the subgame is winning). Thus we have exhibited $g_{\mathcal{L}}$ a winning strategy for $G(\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$. We conclude by using Theorem 3.1. □

B.2 Partial Order Reduction of Unobservable Actions

Partial order reductions allow reducing the state space that needs to be searched in model checking or state exploration algorithms [Peled 1998]. A partial order reduction is based around the idea of commutative actions. If two actions commute, then either order of execution is equivalent. We use two partial order reductions that reduce the state space that Algorithm 1 must explore. They can be used separately or in tandem. One modifies the CFG construction, *POR*-CFG. The other *POR*-TURN reduces the set of strategies we must consider for Verifier.

In weak simulations, it is impossible to observe silent actions. Since threads do not share memory, silent actions from two processes executing in parallel may be reordered. We apply this intuition to reduce the CFG constructed for two processes run in parallel. For two processes T_1 and T_2, the normal construction for $T_1 \| T_2$ is to take the Cartesian product of each process's CFG. From the reduced CFGs of each process, we compute the reduced CFG of $T_1 \| T_2$ by first executing every unobservable action of T_1, then executing every unobservable action of T_2. When T_1 and T_2 both only have observable actions to execute, the construction expands each observable action and repeats the process. For loopy programs, we first compute a *cutset* for both processes' CFG: a set of vertices such that removing them from the graph results in an acyclic graph. We say that an action is *pseudo-observable* if it emanates from the cutset or it is observable. By considering cut-points as pseudo-observable, we ensure that the reduced system is observationally equivalent (weak simulation equivalence) with the original system. Without considering cut-points, if one process executes a non-terminating unobservable loop, then the observable behaviors of the other process would no longer be a behavior of the reduced system.

The second partial order reduction, *POR*-Turn, reduces the set of strategies Algorithm 1 considers. In weak simulations, Verifier may delay its choice until forced to match an observable move. We call the set of strategies that delay Verifiers choice Lazy strategies. When Falsifier plays an unobservable action, Verifier immediately pass. This is always valid for Verifier, if $\sigma_t \overset{\tau}{\Longrightarrow} \sigma'_t$ and $\sigma'_t \overset{a}{\Longrightarrow} \sigma''_t$ then clearly $\sigma_t \overset{a}{\Longrightarrow} \sigma''_t$. Similar to the *POR*-CFG, we begin by computing a cutset for each program's CFG. A move of Falsifier is pseudo-observable if the move is observable or leads to a cutpoint. Whenever Falsifier plays a non pseudo-observable action, then Verifier will always chose to pass. When Falsifier plays a pseudo-observable action then Verifier plays their turn. Verifier's turn lasts until it plays a pseudo-observable action of it's own that matches Falsifier's action (e.g. sends match sends, receives match receives, etc.). Verifier may still choose to pass if Falsifier's move was unobservable.

Theorem B.2. *For any contextual simulation, $\{\mathcal{P}\}\ S \lesssim T\ \{Q\}$, if we apply either (or both) POR-CFG or POR-TURN to Algorithm 1 and Algorithm 1 terminates with a game tree $\mathcal{T}$ then $\{\mathcal{P}\}\ S \lesssim T\ \{Q\}$ is valid. If it returned a counter strategy, then $\{\mathcal{P}\}\ S \lesssim T\ \{Q\}$ is not valid.*

Proof Sketch. The transition system induced by *POR*-CFG, is weakly simulation equivalent to the original transition system—there is a weak simulation in both directions. By Theorem 4.1 we know the algorithm is sound for the reduced game. We can combine the witnessing simulation relation for the contextual simulation and compose it with the weak simulation from the reduced transition system to the full transition system to get a simulation relation witnessing the conclusion. If Falsifier wins the reduced game, then necessarily Falsifier wins the full game.

When the algorithm uses *POR*-Turn and terminates with a strategy tree $\mathcal{T}$, the produced strategy is still a strategy for the full game. Thus by Theorem4.1 the conclusion holds. If Falsifier has a winning strategy, then Falsifier beats Verifier when Verifier plays any lazy strategy. We show that every strategy of Verifier may be reduced to an equivalent lazy strategy (one strategy is winning iff the other wins). Given a verifier strategy g, we commute the corresponding lazy strategy l by delaying g's actions until Falsifier makes an observable turn at which point, l will play each of the delayed actions. Since Falsifier beat every lazy strategy of Verifier, Falsifier can beat every strategy of Verifier. Thus Falsifier must win any play conforming to its strategy. Thus the conclusion holds. □

C PROOFS

THEOREM 2.4. *Let φ be any formula of the universal fragment of action CTL* without next-time operators ($\forall ACTL^*$ -$\{X_p, X_\tau\}$). If program P is related to program Q by a divergence preserving weak simulation and Q satisfies φ then P satisfies φ.*

PROOF OF THEOREM 2.4. We begin by first defining the formal definition of $\forall ACTL^*$ -$\{X_p, X_\tau\}$ and its satisfiability relation that we consider in our proof. Our definition closely follows from [Nicola and Vaandrager 1990].

Action Formulas. Let A be the set of Atomic action predicates. The set of action predicates is defined as the following grammar:

$$F, G \quad ::= a \in A \mid \top \mid \neg F \mid F \wedge G$$

For an action, $\alpha \in \Sigma$ (see Section 2 for definition of Σ), and action formula, F, we use $\alpha \models F$ to denote that α *satisfies* F and $\alpha \not\models F$ that α does not satisfy F. The below rules inductively define when an action formula is satisfiable.

$$\begin{array}{lll} \alpha \models \top & \text{always} & \\ \alpha \models \neg F & \text{iff} & \alpha \not\models F \\ \alpha \models F \wedge G & \text{iff} & \alpha \models F \text{ and } \alpha \models G \end{array}$$

$\forall ACTL^*$ -$\{X_p, X_\tau\}$ Syntax. We define the universal fragment of action CTL^* without next-time operators ($\forall ACTL^*$ -$\{X_p, X_\tau\}$) using the following grammar:

$$\begin{array}{ll} \varphi, \psi & ::= \text{true} \mid \text{false} \mid \varphi \wedge \psi \mid \varphi \vee \psi \\ & \quad \mid \forall\varphi \mid \varphi\ {}_F\mathrm{U}_G\ \psi \mid \varphi\ {}_F\mathrm{U}\ \psi \mid \mathrm{G}\varphi \end{array}$$

Note: We may define the non-modal until operator U and the eventually operator F in terms of the other operators:

$$\varphi\ \mathrm{U}\ \psi \triangleq \varphi\ {}_\top\mathrm{U}\ \psi$$
$$\mathrm{F}\varphi \triangleq \text{true U } \varphi$$

Program Runs. Given a program P and a program state $s \in S_P$ of P, a *path* from s is a (possibly infinite) sequence of transitions, $\pi = \langle s_0, \alpha_0, s_0' \rangle \langle s_1, \alpha_1, s_1' \rangle \in \longrightarrow_P^{\leq\omega}$, beginning from s (i.e. $s_0 = s$ and $\forall i.\ s_i' = s_{i+1}$). A path is *maximal* if it is either infinite or ends in a state with no out-going transitions.

A *run* from $s \in S_P$ is a pair $\rho = \langle s, \pi \rangle$ where π is a path from S. We use $\mathit{first}(\rho)$ to denote s, $\mathit{path}(\rho)$ to denote π. If π is finite, we use $\mathit{last}(\rho)$ to denote the last state of π. We say ρ is *maximal* iff π is maximal.

Given two runs, ρ and θ, such that $\mathit{last}(\rho) = \mathit{first}(\theta)$, we use $\rho\theta$ to represent concatenation (i.e. $\rho\theta = \langle \mathit{first}(\rho), \mathit{path}(\rho)\mathit{path}(\theta) \rangle$).

Given two runs, ρ and θ, we use $\rho < \theta$ and $\rho \leq \theta$ to denote that θ is a proper suffix, respectively a suffix, of ρ. Formally, $\rho < \theta$ iff $\mathit{first}(\theta) = \mathit{last}(\rho)$ and $\rho \leq \theta$ iff there is some ρ', η, and θ' such that $\rho = \rho'\eta$ and $\theta = \eta\theta'$.

Given a program state s, we use $\mu runs(s)$ to denote the set of maximal runs starting from s.

$\forall ACTL^*$ -$\{X_p, X_\tau\}$ Satisfiability. Given a program a run, ρ, of program P and a $\forall ACTL^*$ -$\{X_p, X_\tau\}$ formula φ, we use $\langle \rho, P \rangle \models \varphi$ (or simply $\rho \models \varphi$) to denote that the program state s satisfies the formula φ. A program state $s \in S_P$ of P satisfies φ when $\langle s, \epsilon \rangle \models \varphi$. We say P satisfies φ when every initial state of P satisfies (e.g. $\forall s \in I_P.\ \langle s, \epsilon \rangle \models \varphi$). We define the satisfiability of a $\forall ACTL^*$ -$\{X_p, X_\tau\}$ inductively as follows:

$$
\begin{array}{lll}
\rho \models \text{true} & \text{always} & \\
\rho \models \text{false} & \text{never} & \\
\rho \models \varphi \wedge \psi & \text{iff} & \rho \models \varphi \text{ and } \rho \models \psi \\
\rho \models \varphi \vee \psi & \text{iff} & \rho \models \varphi \text{ or } \rho \models \psi \\
\rho \models \forall\varphi & \text{iff} & \forall \rho' \in \mu\text{runs}(\mathit{first}(\rho)).\ \rho' \models \varphi \\
\rho \models \varphi \ {}_F\mathrm{U}_G\ \psi & \text{iff} & \exists \rho', \theta \text{ s.t. } \begin{cases} \rho = \rho'\theta \\ \theta \models \psi \\ \exists \pi, s, \alpha, s' \text{ s.t. } \begin{cases} \mathit{path}(\rho') = \pi \, \langle s, \alpha, s' \rangle \text{ and } \alpha \models G \text{ and} \\ \forall i, s_i, \alpha_i, s_i'.\ \text{if } \pi_i = s_i, \alpha_i, s_i' \text{ then } \alpha_i = \tau \text{ or } \alpha_i \models F \end{cases} \\ \forall \eta.\ \rho' \leq \eta < \theta \Rightarrow \eta \models \varphi \end{cases} \\
\rho \models \varphi \ {}_F\mathrm{U}\ \psi & \text{iff} & \exists \rho', \theta \text{ s.t. } \begin{cases} \rho = \rho'\theta \\ \theta \models \psi \\ \forall i, s, \alpha, s'.\ \text{if } \mathit{path}(\rho')_i = \langle s, \alpha, s' \rangle \text{ then } \alpha = \tau \text{ or } \alpha \models F \\ \forall \eta.\ \rho' \leq \eta < \theta \Rightarrow \eta \models \varphi \end{cases} \\
\rho \models \mathrm{G}\varphi & \text{iff} & \forall \rho', \theta.\ \text{if } \rho = \rho'\theta \text{ then } \theta \models \varphi
\end{array}
$$

Now that we have formally defined the syntax and satisfiability of $\forall ACTL^*$ -$\{X_p, X_\tau\}$, we may now proceed with the proof that divergence preserving weak simulations preserve satisfiability of $\forall ACTL^*$ -$\{X_p, X_\tau\}$.

Proof. We begin by proving two lemmas.

LEMMA C.1. *Fix a program P. Let φ be any $\forall ACTL^*$ -$\{X_p, X_\tau\}$ formula, ρ be any finite run of P, and θ be any finite and silent run such that $\rho < \theta$. If $\rho \models \varphi$ then $\rho\theta \models \varphi$.*

PROOF. We proceed by induction on $\rho \models \varphi$.

Case true: necessarily $\rho\theta \models$ true.

Case false: the hypothesis $\rho \models$ false is impossible.

Case $\varphi \wedge \psi$:

By assumption $\rho \models \varphi$ and $\rho \models \psi$. By the IH, $\rho\theta \models \varphi$ and $\rho\theta \models \psi$. Thus we can conclude $\rho\theta \models \varphi \wedge \psi$.

Case $\varphi \vee \psi$:

By assumption either $\rho \models \varphi$ or $\rho \models \psi$. By the IH, we either have $\rho\theta \models \varphi$ or $\rho\theta \models \psi$. Thus we can conclude $\rho\theta \models \varphi \vee \psi$.

Case $\forall\varphi$:

By assumption, for ever maximal run ρ' from $\mathit{first}(\rho)$ satisfies φ. Necessarily $\mathit{first}(\rho) = \mathit{first}(\rho\theta)$, thus we may conclude $\rho\theta \models \forall\varphi$.

Case $\varphi \ {}_F\mathrm{U}_G\ \psi$:

By assumption, we know there is some ρ' and θ' such that (1) $\rho = \rho'\theta'$, (2) $\theta' \models \psi$, (3) There is some π, s, α, s' such that $\mathit{path}(\rho') = \pi \, \langle s, \alpha, s' \rangle$ and $\alpha \models G$, and every observable action in π satisfies F, and (4) $\forall \eta . \rho' \leq \eta < \theta' \Rightarrow \eta \models \varphi$.

Let $\theta'' = \theta'\theta$. Clearly $\rho\theta = \rho'\theta''$. By (2) and the IH $\theta'' \models \psi$. Using these facts and (3) and (4), we may conclude $\rho\theta \models \varphi \ {}_F\mathrm{U}_G\ \psi$.

Case $\varphi \ {}_F\mathrm{U}\ \psi$: This case proceeds similarly as the previous case.

Case $\mathrm{G}\varphi$:

By assumption, for every ρ' and θ' such that $\rho = \rho'\theta'$ we have $\theta' \models \varphi$. By the IH we may then assume that $\theta'\theta \models \varphi$. We may then conclude that $\rho\theta \models \mathrm{G}\varphi$.

□

Lemma C.2. *Fix a program P. $\forall ACTL^*$ -$\{X_p, X_\tau\}$ formula, ρ be any finite run of P, and θ be any finite and silent run such that $\theta < \rho$. If $\theta\rho \models \varphi$ then $\rho \models \varphi$.*

Proof. We proceed by induction on $\theta\rho \models \varphi$.
Case true: necessarily $\rho \models$ true.
Case false: the hypothesis $\theta\rho \models$ false is impossible.
Case $\varphi \wedge \psi$:
By assumption $\theta\rho \models \varphi$ and $\theta\rho \models \psi$. By the IH, $\rho \models \varphi$ and $\rho \models \psi$. Thus we can conclude $\rho \models \varphi \wedge \psi$.
Case $\varphi \vee \psi$:
By assumption either $\theta\rho \models \varphi$ or $\theta\rho \models \psi$. By the IH, we either have $\rho \models \varphi$ or $\rho \models \psi$. Thus we can conclude $\rho \models \varphi \vee \psi$.
Case $\forall\varphi$:
Let ρ' be any maximal run from *first*(ρ). $\theta\rho'$ must be a maximal run from *first*$(\theta\rho)$. By assumption we have $\theta\rho' \models \varphi$. Using the IH, we may then show $\rho' \models \varphi$. Thus we may conclude $\rho \models \forall\varphi$.
Case $\varphi\ {}_F\mathrm{U}_G\ \psi$:
By assumption we know there is some transition in $\theta\rho$ that satisfies G. Necessarily, it must appear in ρ, otherwise, θ must not be silent. Let $\rho''\theta'$ be this partition. Since θ is silent, ρ'' must be some $\theta\rho'$. We may equivalently partition ρ into $\rho'\theta'$. Using the IH we may then prove each of the remaining conditions to show $\rho \models \varphi\ {}_F\mathrm{U}_G\ \psi$.
Case $\varphi\ {}_F\mathrm{U}\ \psi$: Either the split of $\theta\rho$ into ρ' and θ' occurs in θ or in ρ. In the first case, we can split ρ into ϵ and ρ and then need only prove $\rho \models \psi$. This may be accomplished using the IH and knowledge that θ' of which ρ is a suffix satisfied ψ. The second case proceeds similarly as the ${}_F\mathrm{U}_G$ case.
Case $\mathrm{G}\varphi$:
By assumption, every suffix of $\theta\rho$ satisfies φ. Clearly, every suffix of ρ must then satisfy φ. Thus we many conclude $\rho \models \mathrm{G}\varphi$.

□

Before proceeding with our main proof. Let P be a program that is divergence preserving weakly simulated by program Q. We define when a run of P is similar to a run of Q (according to simulation relation R). We say the run ρ_P is similar to the run ρ_Q, when ρ_Q is the sequence of transitions witnessing the simulation for each transition in ρ_P. If ρ_P is a maximal run, then so is ρ_Q, and if ρ_P ends in an infinite silent suffix, then ρ_Q's corresponding suffix must be the sequence of transitions witnessing the divergence preserving property.

We now proceed with our main proof of Theorem 2.4. For which we prove the more general case: Let P and Q be programs that are related by the divergence preserving weak simulation R. Let φ be any $\forall ACTL^*$ -$\{X_p, X_\tau\}$ formula, and ρ_P and ρ_Q are runs of P and Q respectively. If $\rho_P R \rho_Q$ and $\rho_Q \models \varphi$ then $\rho_P \models \varphi$.

We proceed by induction on $\rho_Q \models \varphi$.
Case true: necessarily $\rho_P \models$ true.
Case false: the hypothesis $\rho_Q \models$ false is impossible.
Case $\varphi \wedge \psi$:
By assumption $\rho_Q \models \varphi$ and $\rho_Q \models \psi$. By the IH, $\rho_P \models \varphi$ and $\rho_P \models \psi$. Thus $\rho_P \models \varphi \wedge \psi$.
Case $\varphi \vee \psi$:
By assumption, either $\rho_Q \models \varphi$ or $\rho_Q \models \psi$. By the IH, either $\rho_P \models \varphi$ or $\rho_P \models \psi$. Thus $\rho_P \models \varphi \vee \psi$.
Case $\forall\varphi$:

Let ρ'_P be any maximal run from $\mathit{first}(\rho_P)$. We construct a new run ρ'_Q that is R-related to ρ'_P. For each transition of ρ'_P we concatenate the transitions that witness the simulation property's observational equivalence condition. If ρ'_P has an infinite silent suffix, then we match each transition of the suffix using the transitions witnessing the divergence preserving property for the suffix. By construction $\rho'_P R \rho'_Q$. Necessarily, ρ'_Q is also maximal. By assumption, $\rho'_Q \models \varphi$. By the IH, $\rho'_P \models \varphi$ and thus $\rho_P \models \forall\varphi$.

Case $\varphi\ {}_F\mathrm{U}_G\ \psi$:

Clearly, ρ_P and ρ_Q must be observationally equivalent. Since we know there is some transition in ρ_Q that satisfies G, there must be a transition of ρ_P that similarly satisfies G. We partition ρ_P into $\rho'_P\theta_P$ at this transition. We now partition ρ_Q into $\rho'_Q\rho\theta_Q$ such that ρ'_Q is R-related to ρ'_P, similarly for θ_Q and θ_P. And ρ is the sequence of transitions witnessing the observationally equivalent sequence of transitions to the transition in ρ_P that satisfies G. Since every transition of ρ'_Q must either be silent or satisfy F, we may conclude that every transition of ρ'_P holds similarly. Let ρ' and θ be the partition of ρ that splits on the transition that satisfies G. For each $\rho'_P \leq \eta_P < \theta_P$, we can use the fact that there is some $\rho'_Q \leq \eta_Q \leq \theta_Q$ such that $\eta_P R \eta_Q$ and $\eta_Q \models \varphi$. By the IH, it must be that $\eta_P \models \varphi$. We additionally know that $\rho' \models \varphi$. We use Lemma C.1 to show that $\rho \models \varphi$. Using the IH, we may conclude that the transition of ρ_P satisfying G must also satisfy φ. We then use Lemma C.2, the IH, and the assumption that $\theta'\theta_Q \models \psi$ to conclude that $\theta_P \models \psi$. Thus we may conclude that $\rho_P \models \varphi\ {}_F\mathrm{U}_G\ \psi$.

Case $\varphi\ {}_F\mathrm{U}\ \psi$: This case proceeds similarly as the preceding case.

Case $\mathrm{G}\varphi$:

Let ρ'_P be any suffix of ρ_P. We denote with ρ'_Q the suffix of ρ_Q such that $\rho'_P R \rho'_Q$. Since ρ'_Q is a suffix of ρ_Q we know that $\rho'_Q \models \varphi$. We use the IH to prove $\rho'_P \models \varphi$. Thus we may conclude $\rho_P \models \mathrm{G}\varphi$.

□

Theorem 3.1. *The contextual simulation $\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$ is valid if and only if Verifier has a winning strategy for $\mathcal{G}(\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\})$.*

Proof of Theorem 3.1.

By assumption, *src* and *tgt* are over disjoint variables say X and Y. Given a valuation $\lambda : X \cup Y \to \mathbb{Z}$, we use $\lambda|_X$ to denote the valuation equivalent to λ restricted to the variables in X and analogously for $\lambda|_Y$.

Case $\Rightarrow$:

Let R be the weak simulation relation witnessing $\models \{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$.

We now construct Verifier's strategy g_R. Let $s = s_0s_1...s_n$ be any position conforming to g_R. We begin by induction on n to show that if s conforms to g_R and Falsifier has not made an illegal move then if s_n is a Verifier move then Verifier has a legal response $g_R(s)$; otherwise, if $s_n = F\left\langle l_{src}, l_{tgt}, \lambda\right\rangle$ then $(l_{src} \triangleright \lambda|_X)R(l_{tgt} \triangleright \lambda|_Y)$ or ($l_{src} = out_{src}$ and $l_{tgt} = out_{tgt}$ and $[\![Q]\!]_\lambda$ is true.

Case $n = 0$:

By the initialization rule, Falsifier must choose a Falsifier place $F\left\langle in_{src}, in_{tgt}, \lambda\right\rangle$ such that $[\![\mathcal{P}]\!]_\lambda$ is true. By definition of weak simulation, necessarily $(\lambda|_X \triangleright in_{src})R(\lambda|_Y \triangleright in_{tgt})$.

Case $n = n' + 1$:

Let $i \leq n'$ be the greatest index such that $s_i = F\left\langle l_{src}, l_{tgt}, \lambda\right\rangle$ is a Falsifier place. By the inductive hypothesis, $\lambda|_X \triangleright l_{src}$ is R-related to $\lambda|_Y \triangleright l_{tgt}$. By assumption s_{i+1} is a legal move and for every $i+1 < k \leq n$, s_k conforms to g_R. Since s_{i+1} is legal, it must be some $V\left\langle \alpha, l'_{src}, l_{tgt}, \lambda'\right\rangle$ such that $(\lambda \triangleright l_{src}) \xrightarrow{\alpha}_{src} (\lambda' \triangleright l'_{src})$ (or $l_{src} = l'_{src} = out_{src}$, $\alpha = \tau$ and $\lambda' = \lambda$).

Because S and T are over disjoint variables, clearly $(\lambda|_X \triangleright l_{src}) \xrightarrow{\alpha}_{src} (\lambda' \triangleright l'_{src})$ and $\lambda|_Y = \lambda'|_Y$.
By the definition of weak simulation there must be some sequence of transitions that witness $(\lambda|_y \triangleright l_{tgt}) \overset{\alpha}{\Longrightarrow}_{tgt} (\lambda_{tgt} \triangleright l'_{tgt})$ where $\lambda'|_X \triangleright l_{src}$ is R-related to $\lambda_{tgt} \triangleright l'_{tgt}$ (or $l'_{src} = out_{src}$ and $l_{tgt} = out_{tgt}$ and $\lambda'|_X \uplus \lambda_{tgt}$ satisfies Q).
W.l.o.g. assume we always pick the same sequence of transitions if multiple such transitions exist.
Let $(\lambda_0 \triangleright l_0) \xrightarrow{\beta_1}_{tgt} \dots \xrightarrow{\beta_m}_{tgt} (\lambda_m \triangleright l_m)$ be this sequence, where $\lambda_0 = \lambda|_y$, $l_0 = l_{tgt}$, $\lambda_m = \lambda_{tgt}$, and $ł_m = l'_{tgt}$.
Let α_j be τ if for any $j' \le j$ β_j is α, otherwise let α_j be α.
Note, by the definition of weak simulation β_j is either τ or α (if $\alpha \neq \tau$ then exactly one β_j is α). Thus, α_m must be τ.
For each $1 \le j \le m$, our strategy chooses s_{i+j} to be $V\left\langle \alpha_j, l'_s, l_j, \lambda'|_X \uplus \lambda_j \right\rangle$ $(g_r(s_0 \dots s_{i+j}) = V\left\langle \alpha_j, l'_s, l_j, \lambda'|_x \uplus \lambda_j \right\rangle)$.
For s_{i+m+1} our strategy chooses $F\left\langle l'_{src}, l'_{tgt}, \lambda'|_X \uplus \lambda_{tgt} \right\rangle$ $(g_r(s_0 \dots s_{i+m+1}) = F\left\langle l'_{src}, l'_{tgt}, \lambda'|_X \uplus \lambda'_{tgt} \right\rangle)$.
By our assumption, Falsifier has not made an illegal move and every move chosen by Verifier conforms to g_r.
Thus every move $s_0, \dots, s_i$ must be legal (by assumption and the inductive hypothesis).
For each $i + 1 < k \le i + m + 1$, $s_0 \dots s_k$ is a legal position. Each Verifier choice from $i + 2$ to $i + m + 1$ is legal.
Necessarily $n \le i + m + 1$, otherwise i was not the greatest index of a Falsifier node in $s_0 \dots s_n$. Clearly if $n < i + m + 1$, we may conclude that Verifier has a legal move (i.e. s_{n+1}). If $n = i + m + 1$, then necessarily $\lambda'|'_X \triangleright l'_{src}$ is R-related to $\lambda_{tgt} \triangleright l_{tgt}$ or $l_s = out_{src}$ and $l_{tgt} = out_{tgt}$ and $[\![Q]\!]_{\lambda'|_X \uplus \lambda_{tgt}}$ is true.

We have proven the Lemma. Let p be any play that conforms to g_R. By the above lemma, none of the three winning conditions for Falsifier are possible. Thus p is won by Verifier and g_R is a winning strategy.

Case $\Leftarrow$:

Let g be Verifier's winning strategy for the game $\mathcal{G}(\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\})$. For any play p that conforms to g, let $R_p = \{(\lambda|_X \triangleright l_{src}, \lambda|_Y \triangleright l_{tgt}) : p' \cdot F\left\langle l_{src}, l_{tgt}, \lambda \right\rangle$ is a legal prefix of $p\}$, relate source and target program states associated to any Falsifier place in the play whose prefix is legal.
Let $R_g = \bigcup_p R_p$ be the union of every R_p for every play p that conforms to g.
R_g is a divergent preserving weak simulation relation witnessing $\models \{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$.

□

Theorem 3.6. *If there is a well-labeled complete simulation game tree for $\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$, then Verifier has a winning strategy for $\mathcal{G}(\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\})$.*

Proof of Theorem 3.6.
Let $\mathcal{L} = \langle\langle F, V, E, r, L, S, T\rangle, \Phi, K, G, X, \triangleright, m\rangle$ be any complete well-labeled simulation game tree for $\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\}$.
We formalize *Places* in Section 3 using $F\text{-}pred^*_e$ as defined in Section 4 and *letter* which takes the output of $F\text{-}pred^*_e$ (a send or receive command or None) and a valuation and computes a letter associated to the command (or τ for None).

$$letter(\texttt{send } m \texttt{ chan}(c), \lambda) = \mathbf{s}(\llbracket m \rrbracket_\lambda, \llbracket c \rrbracket_\lambda)$$
$$letter(\texttt{receive } x \texttt{ chan}(c), \lambda) = \mathbf{r}(\llbracket x \rrbracket_\lambda, \llbracket c \rrbracket_\lambda)$$
$$letter(\text{None}, \lambda) = \tau$$

$$Places(n) = \begin{cases} \{F \langle S(n), T(n), \lambda \rangle : \llbracket \Phi(n) \rrbracket_\lambda \text{ is true}\} & \text{if } n \in F \\ \left\{V \left\langle letter(F\text{-}pred_e^*(n), \lambda), S(n), T(n), \lambda \right\rangle : \llbracket \Phi(n) \rrbracket_\lambda \text{ is true}\right\} & \text{if } n \in V \end{cases}$$

We now Formalize g the strategy defined by *Places* (as described in Section 3).
Let $p \cdot m$ be any position ending in a Verifier place ($m = V \left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle$).
If m is not associated to any node ($m \notin Places(n)$ for any node n). Then let $g(p \cdot m)$ be $F \left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle$.
Otherwise, let n be any node m is associated with (i.e. $m \in Places(n)$). By definition of *Places*, n must be a V-node.
Let n' be any successor of n such that the valuation of m satisfies the guard of the edge from n to n' (i.e. $\llbracket G(n, n') \rrbracket_\lambda$ is true).
If $L(n, n')$ is observable, then let $g(p \cdot m)$ be $V \langle \tau, \ell_{src}, T(n'), \lambda' \rangle$, where $(\lambda \triangleright \ell_{tgt}) \xrightarrow{\alpha}_{tgt} (\lambda' \triangleright T(n'))$.
Note: by the consecution constraint, exactly one such transition of this form must exist.
If $L(n, n')$ is unobservable and not a havoc command, then let $g(p \cdot m)$ be $V \langle \alpha, \ell_{src}, T(n'), \lambda' \rangle$, where $(\lambda \triangleright \ell_{tgt}) \xrightarrow{\tau}_{tgt} (\lambda' \triangleright T(n'))$.
Note: by the consecution constraint, exactly one such transition of this form must exist.
If $L(n, n')$ is `havoc x. b`, then let $g(p \cdot m)$ be $V \langle \alpha, \ell_{src}, T(n'), \lambda[x \mapsto c] \rangle$, where c is $\llbracket K(n, n') \rrbracket_\lambda$.
Note: by the consecution constraint, $(\lambda \triangleright \ell_{tgt}) \xrightarrow{\tau}_{tgt} (\lambda[x \mapsto c] \triangleright T(n'))$.

We have finished defining g. We now proceed to prove that g is a winning strategy for $\mathcal{G}(\{\mathcal{P}\}\ src \lesssim tgt\ \{Q\})$.
Let $p = m_0 m_1 \ldots$ be any play conforming to g.
We prove by induction (over prefixes of p), that Verifier does not make the first illegal move. We additionally prove that if the prefix $m_0 \ldots m_n$ is legal then m_n is associated to a node (or $m_n = V \langle \tau, \ell_s, \ell_t, \lambda \rangle$ and $F \langle \ell_s, \ell_t, \lambda \rangle$ is associated to a node). and the node associated with m_n is the successor of the node associated with m_{n-1}.

Case m_0:
The first move is made by Falsifier, thus Verifier has not yet made an illegal move. For m_0 to be legal it must take the form $F \left\langle in_{src}, in_{tgt}, \lambda \right\rangle$ where λ satisfies $\mathcal{P}$. By the initial constraint, m_0 must be associated to r.

Case $m_0 \ldots m_n m_{n+1}$:
By the inductive hypothesis, Verifier did not make the first illegal move of the prefix $m_0 \ldots m_n$. If $m_0 \ldots m_n$ is not legal, then Falsifier must have made the first illegal move. And we have proved the lemma.
Otherwise $m_0 \ldots m_n$ is legal.

Case $m_n = F \left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle$:
m_{n+1} was chosen by Falsifier, thus Verifier has not yet made an illegal move. If m_{n+1} is legal then it must take the form $V \left\langle \alpha, \ell'_{src}, \ell_{tgt}, \lambda' \right\rangle$ where $(\lambda \triangleright \ell_{src}) \xrightarrow{\alpha}_{src} (\lambda' \triangleright \ell'_{src})$. Since m_n is associated with some F-node u, by the adequacy and consecution constraints, there must be some successor of v such that if $v \in V$ then $m_{n+1} \in Places(v)$; otherwise, $\alpha = \tau$ and $F \left\langle \ell'_{src}, \ell_{tgt}, \lambda' \right\rangle \in Places(v)$

Case $m_n = V \left\langle \alpha, \ell_{src}, \ell_{tgt}, \lambda \right\rangle$:
m_{n+1} must be $g(m_0 \ldots m_n)$. Either m_n is associated to some node u or it is not. If it is not, then by the inductive hypothesis α is τ and $m_{n+1} = F \left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle$. By the consecution rules,

$F\left\langle \ell_{src}, \ell_{tgt}, \lambda \right\rangle$ must be associated to a node v. (m_{n-1} must be associated to some node, and m_{n+1} is associated with the chosen successor based on m_n).
If m_n is associated to some node u, then m_{n+1} must be $g(m_0...m_n)$. As described above, by the consecution rules m_{n+1} must be a legal move. m_{n+1} was computed by selecting some successor node v of u such that $G(u, v)$ is satisfied by λ. Either m_{n+1} is associated with v (if $v \in V$) or it is not and the corresponding Verifier move is associated with v. By this point we can be assured m_{n+1}'s letter to match must be τ, due to the observational matching constraint.

We have now proven that for any conforming play p Verifier has not made an illegal move. Thus Falsifier cannot win the play by forcing Verifier to make an illegal move. The second win condition of Falsifier is ruled out by the well-labeledness's final constraint. The third win condition of Falsifier is also ruled out by the well-foundedness constraints: for there to be an infinite sequence where Verifier always passes or always continues, there must be a non well-founded cycle of F-nodes or V-nodes respectively.
Thus g is a winning strategy for Verifier of $\mathcal{G}(\{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\})$. □

Theorem 4.1. *Algorithm 1 is sound. For any contextual simulation, if* Strategy-synthesis*($\{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$) terminates with a simulation strategy, then $\models \{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$. If* Strategy-synthesis *instead terminates with a simulation counter-strategy then $\not\models \{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$.*

Proof of Theorem 4.1. The algorithm maintains the invariant that $\mathcal{L}$ is well-labeled. If the algorithm terminates with a strategy $\mathcal{L}$ then we know the unwinding is complete and well-labeled. Thus, we may then use Theorems 3.6 and 3.1 to conclude that $\models \{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$. Algorithm 1 terminates with a counter strategy, then Falsifier has a winning strategy, and so by Theorem 3.1 we may conclude that $\not\models \{\mathcal{P}\}\ src \lesssim tgt\ \{\mathcal{Q}\}$. □